Object Oriented Programming with Smalltalk

by Bryce Hendrix

Disclaimer: This document is provided for educational purposes, and contains some non-original content. If you are the author of any material included in this document and would like it removed, please send mail to Bryce Hendrix at ***hendrix@engr.arizona.edu*** and it will be removed immediately. Contents of this document cannot be used for commercial usage without explicit permission from the author.

Lecture 1 : What is an Object?

- **2 Rules of Smalltalk**
  - *Everything* is an object
  - Objects respond only to messages
  - Ex: Automobile object
    - Variables: velocity, weight, and color
    - Methods: accelerate and decelerate
- **What's special about an Object?**
  - Objects contain both *state* and *behavior* and communicate with one another via *messages*.
  - Automobile's velocity variable is changed by accelerate method
  - An application is a group of objects interacting in a coordinate fashion
  - Stop light application manages many Automobile objects
- **OO versus Procedural Approach to programming**
  - Aspects of Procedural Approach
    - Behavior is vested in the procedures
    - Procedures must know data structures
    - Procedures communicate only via data
  - Procedural approach places too much emphasis on data, rather than the behavior of the application.
  - Ex: Baker Procedural approach to baking cookies
    - Has 2 structures: Baker structure and cookie structure
    - Steps:
      - Make pointer to a cookie struct
      - Call bakeCookies(Baker, cookie), returns pointer to cookies
    - Review aspects of Procedural Approach
    - Aspects of OO Approach
      - An application is a set of objects communicating via messages
      - An Object's functionality is described by its methods
      - Data required to support an object's functionality is stored in private variables
    - Examples:
      - Baker Object
        - State
          - Weight
          - Height
          - Name
        - Method
          - `bakeCake()`
          - `bakeCookies()`
      - Kitchen Application
        - Objects
          - Baker
          - Chef
          - Dishwasher
          - Oven
        - Procedural Approach
          - Treat each object as a data structure. Each object must have its own data structure & variables.
          - Write a function `wash()`. Note that 4 different functions must be written
        - OO Approach

- Treat each object as object. Objects can inherit variables form each other.
- Write a method `wash()` that operates for all objects. (show in Smalltalk & C)
  - Good Exercise for students: Use polymorphism for one object to do wash methods for Plates object and Cup object
  - Good Exercise for students: write KitchenObject class.

```
KitchenObject subclass: #Baker
	instanceVariableNames:'name weight height'
	classVariableNames:''
	PoolDictionaries:''
	category:''

	name
		^name

	name: aNewName
		name := aNewName

	bakeCake: ingredients
		| cake |
		cake := Cake new from: ingredients.
		^cake

	wash: dirtyDishes
		dirtyDishes := dirtyDishes soak.
		dirtyDishes := dirtyDishes scrub.
		dirtyDishes := dirtyDishes dry.
		^dirtyDishes

KitchenObject subclass: #Dishwasher
	instanceVariableNames:'name weight height'
	classVariableNames:''
	PoolDictionaries:''
	category:''

	name
		^name

	name: aNewName
		name := aNewName

	wash: dirtyDishes
		^ self runCycleOn: dirtyDishes

	runCycleOn: someDishes
		someDishes := someDishes rinse.
		someDishes := dry.
```

```
class CKitchenObject : public CBaker {
public:
	char* name;
	int weight;
	int weight;

public:
	CCake bakeCake(CIngredients ingredients);
	CDishes wash(CDishes dirtyDishes);
}
```

```
CCake CBaker::bakeCake(CIngredients ingredients)
{
	CCake cake = new CCake(ingredients);
	return cake;
}

CDishes CBaker::wash(CDishes dirtyDishes)
{
	dirtyDishes.soak();
	dirtyDishes.scrub();
	dirtyDishes.dry();
	return dirtyDishes;
}

class CKitchenObject : public CDishwasher {
public:
	char* name;
	int weight;
	int height;

public:
	CDishes wash(CDishes dirtyDishes);
}

CDishes CDishwasher::wash(CDishes dirtyDishes)
{
	this.runCycleOn(dirtyDishes);
}

CDishes CDishwasher::runCycleOn(CDishes dirtyDishes)
{
	dirtyDishes.rinse();
	dirtyDishes.dry();
}
```

```
typedef struct {
	char* name;
	int height;
	int weight;
} Worker;

Worker baker;
Worker dishwasher;

char* getName (Worker aWorker)
{
	return aWorker.name;
}

void nameWorker( Worker aWorker, char* newName)
{
	strcpy(aWorker.name, newName);
}

void bakeCake(Ingredient_struct* ingredients,
		Cake* newCake)
{
	newCake = doSomethingWith(ingredients);
}

void bakerWashDishes(Dishes* dirtyDishes)
{
```

```
	soak(dirtyDishes);
	scrub(dirtyDishes);
	dry(dirtyDishes);
}

void dishwasherWashDishes(Dishes* dirtyDishes)
{
	dirtyDishes = runCycleOn(dirtyDishes);
}

void runCycleOn(Dishes* dirtyDishes)
{
	dirtyDishes = rinse(dirtyDishes);
	dirtyDishes = dry(dirtyDishes);
}
```

Lecture 2: Classes and Instances

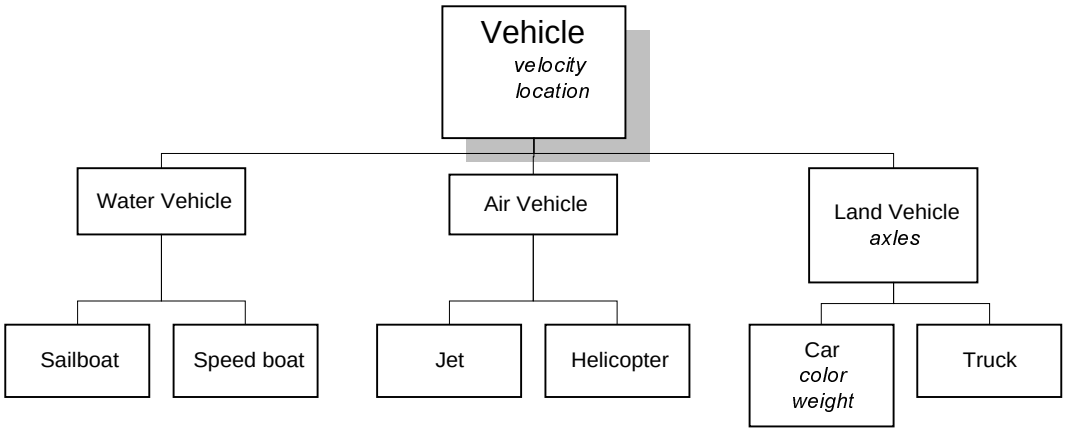
- **Class**
  - A template for objects that share common characteristics.
  - Includes an object's state variable and methods
    - Ex: Vehicle class

Vehicle
Velocity
Location
Color
Weight
Start( )
Stop( )
Accelerate( )

- **Instance**
  - A particular occurrence of an object is defined by a class
    - Classes are sometimes thought of as factories. If we had an automobile factory, the class would be the factory and the automobiles would be the instances of that factory.
  - Each instance has its own values for instance variables
    - Each automobile has its own engine, hood, doors, etc.
  - All instances of a class share the same method
    - Methods are the functions that are applicable to all instances of a class.
      - The method `accelerate` is applicable to all automobiles
    - Ex: A road contains many instances of vehicles, all different colors, going different speeds, starting, stopping, accelerating, etc
    - Ex: cars on a road. It is important to note that car1 and car 3 are not the same object, but are the both instances of the class Car. car1 and car3 are equivalent, but not equal. Equality implies they are the same object.

```
aRoad = Road new.
car1 = Car new withColor: red withSpeed 30.
car2 = Car new withColor: blue withSpeed 45.
car3 = Car new withColor: red withSpeed 30.
```

- **Class Hierarchy**
  - Allows sharing of state and behavior
    - Subclasses are able to use the methods and variables of the parent classes.
  - Each class refines / specializes its ancestors
  - Child can add new state information
    - A Land Vehicle adds the state information regarding the number of axles
  - Child can add, extend or override parent behavior
    - All Vehicles can be driven, but all types of vehicles require different sets of methods to drive
  - Superclass is the parent and subclass is a child
  - Abstract class holds common behavior & characteristics, concrete classes contain complete characterization of actual objects
    - In the Vehicle example, Vehicle is the Superclass, and Sailboat, Speed boar, Jet, Helicopter, Car and Truck are the concrete classes.



Ex: Vehicle hierarchy (leaves are concrete, all other are abstract)

Lecture 3: Messages, Methods, and Programming in Smalltalk

- **Messages**
  - A message specifies what behavior an object is to perform
  - Only way to communicate with an object
  - Implementation is left up to the receiver object
    - Ex: Ask the baker to bake a cake. We don't care how he does it.
    - Ex: `baker bakeCake.`
  - State Information can only be accessed via messages
    - Ex: I want to know how old you are (one of your state variables), so I ask you. I don't care how you compute your age, all I care about is the answer.
    - Ex: `baker age.`
  - The receiver object always returns a result (object).
    - A lot of the time a receiver is modified and it doesn't make sense to return something, so the argument is returned
    - Ex: `#(a b c) at: 3 put: #d` returns #d
- **Methods**
  - A method specifies how a receiver object performs a behavior.
  - Executed in response to a message
  - Must have access to data (must be passed, or contained in object)
    - If there is no access passed or contained in the object, what can be done?
  - Needs detailed knowledge of data
  - Can manipulate data directly
    - Can modify instance variables of the object receiving the message
    - Ex: `#(a b c) at: 3 put #d`. modifies the collection which is the instance variable
  - Returns an object as a result of its execution
    - Since a method is executed in response to a message, and we have already said all messages return an object, it should only make sense that the method returns an object as the result of its execution
  - Has same name as the message name
    - Ex: `#(a b c) size`. `size` is the message called by the receiver, and the `size` method is the method in class `Array` to be executed
  - Visual Works does no type checking on arguments, although the types should be type-compatible.
  - Method returns the receiver object by default, unless explicitly returned
    - Ex: Bob is asked to bake a cake. Bob's 'bake' method explicitly says to return a cake, rather than returning himself to the requester.
    - Ex: the `at:` method of class Interval
      - Explicitly returns a temp variable

```
at: anInteger
        "Answer the number at index position
         anInteger in the receiver interval."
    | answer |
    anInteger > 0
        ifTrue: [
            answer := beginning + (increment *(anInteger
                        - 1)).
            (increment < 0
                and: [answer between: end and:
                        beginning])
                    ifTrue: [^answer].
            (increment > 0
                and: [answer between: beginning and: end])
                    ifTrue: [^answer]].
```

```
            ^self errorInBounds: anInteger
```

- Ex: the `asString` method of class String
  - Returns the receiver (`self`)

```
asString
    "Answer the string representing the
      receiver (the receiver itself)."
    ^self
```

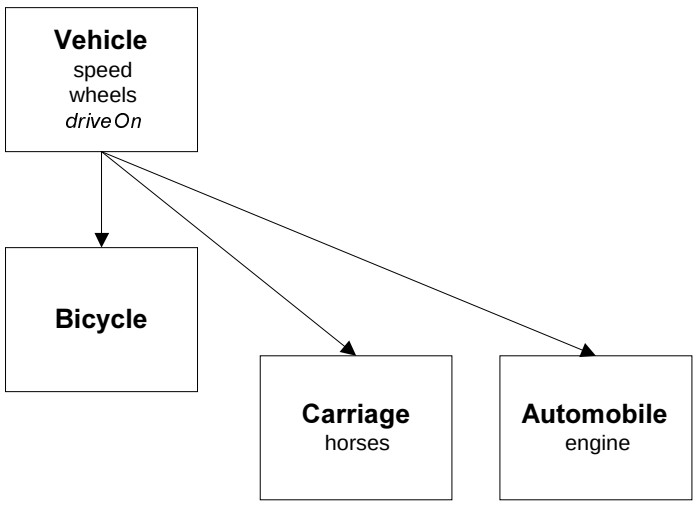
- **Programming in Smalltalk**
  - Code is written and tested in small pieces
    - Usually each method is tested after completion
  - Smalltalk is interpreted
    - Code is compiled into bytecode incrementally during development
      - Once the code has been written, it is “accepted” and compiled into bytecode, then tested.
    - Bytecode is interpreted by the Virtual Machine.
    - The advantage to a Virtual Machine is that different machines can have their own VM to interpret the bytecode. Thus, compiled code should be platform independent.
    - Rather than compile all classes for each program, Smalltalk compiles all of the classes and methods into an “image”

Lecture 4: OO Classification Techniques

- The Vehicle Class Description

```
Object subclass: #Vehicle
       instanceVariableNames: 'speed wheels'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       withWeels: numberOfWheels goingSpeed: aSpeed
              "Creates a new Vehicle Object"
              | aNewVehicle |
              aNewVehicle := self new.
              aNewVehicle wheels := numberOfWheels.
              aNewVehicle speed := aSpeed.
              ^aNewVehicle.


       driveOn: aRoad
              "Returns the reciever, does the driving"
              self speed < aRoad speedLimit
                     ifTrue:
                       [self speed := (self speed) + 1.
                            ^self]
                     ifFalse:
                       [self speed := (self speed) - 1.
                            ^self].
```

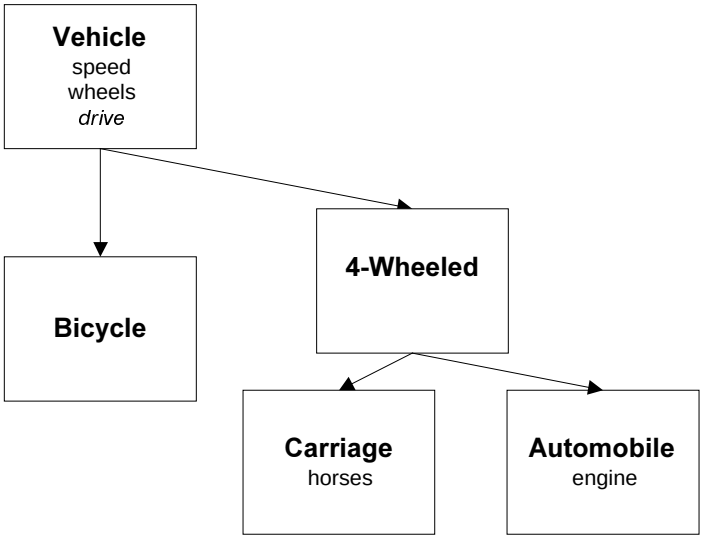
- **Specialization**
  - The act of creating a subclass of class. The new class inherits, overrides, and extends the behavior of the superclass.
  - How?
    - Add instance variables as needed
    - Add, extend, or override methods as needed
  - "is-a" relationship. An automobile "is-a" vehicle.
  - Benefit- code reuse
  - Ex: Class Vehicle exists before Class Automobile is invented. Class Automobile is invented, but based on the methods and variables of Class Vehicle.



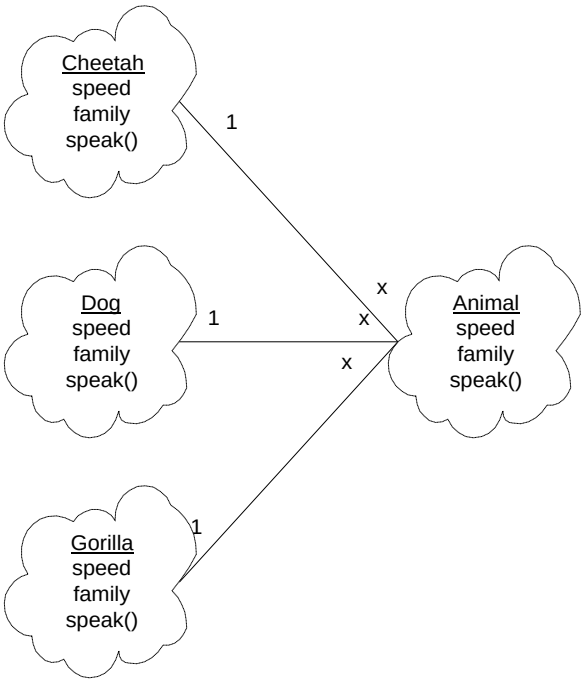

```
Vehicle subclass: #Automobile
       instanceVariableNames: 'speed wheels engine'
```

```
        classVariableNames: ''
        PoolDictionaries: ''
        category: ''.

        withWheels: numberOfWheels goingSpeed: aSpeed withEngine: anEngine
               "Create a new Automobile Object"
               | aNewAuto |
               aNewAuto := self new.
               aNewAuto := Vehicle withWheels: numberOfWheels
                             goingSpeed: aSpeed.
               aNewAuto engine := anEngine.
               ^aNewAuto.

        driveOn: aRoad
               "Returns the receiver, does the driving"
               self speed < aRoad speedLimit
                      ifTrue: [self engine accelerate. ^self]
                      ifFalse: [self engine decelerate. ^self].
```

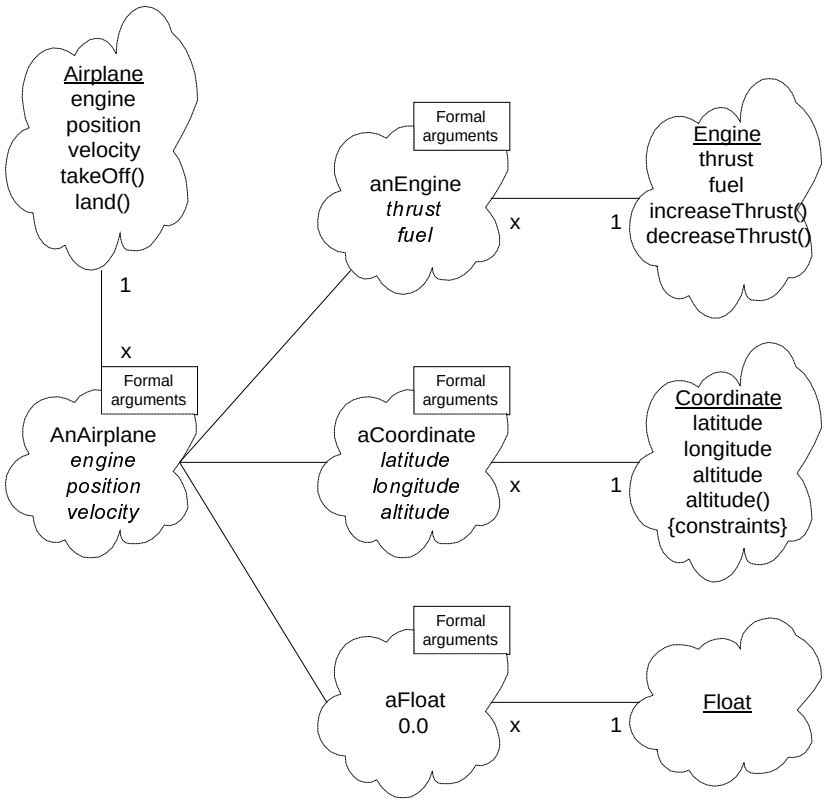
- **Abstraction**
  - The act of creating a superclass for several classes.
  - How?
    - Identify the shared state and /or behavior across the classes
    - Move shared properties to the new abstract superclass
    - Interpose the new abstract superclass in the class hierarchy
  - Benefit: code reuse, simplify maintenance, better understanding
  - Example



  - Example:



- **Composition**
  - The act of creating a class that is composed of instances of other classes (via instance variables). The new class does not inherit form the other classes, but can access their state and behavior via messages.
  - How?
    - Create a new class that is composed of other classes
      - Attributes of the new class are instances of other classes
      - The new class obtains the behavior of composition classes by sending messages to the referenced instances ("delegation").
  - Benefit: provides protection from changes in referenced classes.
  - Behavior is not inherited
  - "has-a" relationship (also known as "is-part-of").
  - Ex: An instance of class Airplane might be composed of instances of the class variables Engine, Position, and Velocity.
    - Engine points to an instance of class Engine (user defined)
    - Position points to an instance of class Position (user defined)
    - Velocity points to an instance of Visual Works system class, Float.
    - An Airplane "has-a" Engine, Position, and Velocity.



- **Factorization**
  - The act of breaking a class into smaller classes.
  - How?
    - Factor the class into smaller classes
      - Create a new class for each distinct type of state / behavior
      - Recombine the new classes via inheritance and /or composition to achieve original functionality.
  - Example: break the class Animal into different Species Classes
  - Benefit: Potential reusable, smaller classes

```
Object subclass: #Vehicle
       instanceVariableNames: 'speed wheels'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       withWeels: numberOfWheels goingSpeed: aSpeed
              "Creates a new Vehicle Object"
              | aNewVehicle |
              aNewVehicle := self new.
              aNewVehicle wheels := numberOfWheels.
              aNewVehicle speed := aSpeed.
              ^aNewVehicle.


       driveOn: aRoad
              "Returns the reciever, does the driving"
              self speed > aRoad speedLimit
                     ifTrue:
                       [self speed := (self speed) + 1. ^self]
                     ifFalse:
```

```
                    [self speed := (self speed) - 1. ^self].


#Vehicle subclass: #TwoWheel
       instanceVariableNames: 'speed wheels balance'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       withWeels: numberOfWheels goingSpeed: aSpeed
              "Creates a new Vehicle Object"
              | aNewVehicle |
              aNewVehicle := self new.
              aNewVehicle wheels := 2.
              aNewVehicle speed := aSpeed.
              ^aNewVehicle.


       driveOn: aRoad
              "Returns the reciever, does the driving"
              self balnce = nil
                     ifTrue: [self speed := 0. ^self].
              self speed < aRoad speedLimit
                     ifTrue:
                       [self speed := (self speed) + 1. ^self]
                     ifFalse:
                       [self speed := (self speed) - 1. ^self].

#Vehicle subclass: #FourWheel
       instanceVariableNames: 'speed wheels fourWheelDrive'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       withWeels: numberOfWheels goingSpeed: aSpeed
                           isFourWheelDrive: anAnswer
              "Creates a new Vehicle Object"
              | aNewVehicle |
              aNewVehicle := self new.
              aNewVehicle wheels := 4.
              aNewVehicle speed := aSpeed.
              aNewVehicle fourWheedDrive := anAnswer.
              ^aNewVehicle.


       driveOn: aRoad
              "Returns the reciever, does the driving"
              self speed < aRoad speedLimit
                     ifTrue:
                       [self speed := (self speed) + 1. ^self]
                     ifFalse:
                       [self speed := (self speed) - 1. ^self].
```

Lecture 5: Encapsulation & Polymorphism

- **Encapsulation**
  - Objects encapsulates *State* as a collection of variables
    - Common practice is to provide a set of private methods for manipulating variables.
    - Example: Baker has work state (ie rolling dough, baking, resting)
      - `baker state.` Returns the baker's state
      - `baker state: 'baking'.` Sets the baker's state
    - Example: The class Engine
      - In the previous lecture we looked the the Automobile class. When we created an instance of the class Automobile, we assumed the instance creation was called with an instance of Engine as an argument
      - An engine must have many private methods. When you turn the ignition, you don't have to start each component of the engine individually. Lets look at a simple engine class

```
Object subclass: #Engine
       instanceVariableNames: 'state pistons battery'
       classVariableNames: ''
       PoolDictionaries: ''
       category: ''.

       start
              "Starts up the engine"
              self startEachComponent.
              ^status.

private
       startEachComponent
              "Checks to see if the battery is charged, and
                     tries to start the pistons"
              status := true.
              pistons := Pistons new.
              battery := Battery new.
              battery status
                     ifFalse: [status := false].
              pistons start
                     ifFalse: [status := false].
```

  - Objects encapsulates *Behavior* as methods invoked by messages
    - Set of methods encompasses everything the object knows how to do
    - Ex: Baker has `setState` method to set `stateVariable`, and `queryState` to get `stateVariable`'s value:

```
setState: aValue
       stateVariable=aValue.

queryState
       ^StateVariable.

Baker Bob do: 'resting'.
Bob queryState.
```

  - Encapsulation protects the state information of an object
    - Legal Example: Baker object can access thoughts (read and write)
    - Illegal Example: Someone else cannot read the baker's thoughts.
  - Encapsulation hides implementation details

    - Don’t care how baker bakes cake.
  - Encapsulation provides a uniform interface for communicating with an object.
    - We can ask the baker to bake a cake, or we can ask the chef to bake a cake. They will do it differently, but we can ask them the same way.
  - Facilitates modularity, code reuse and maintenance.
    - Side note: C++ faq claims encapsulation does not facilitate code-reuse, this is an important difference in the language C++ programmers should consider.
- **Polymorphism**
  - Variety of objects in an application that exhibit the same generic behavior, but implement it differently
    - Ex: Ask a dog to speak, it barks. Ask a cat to speak, it meows. Each animal can be asked to speak, and each will do it differently.
    - Ex: The + operator for class Float and class Integer
      - Float:

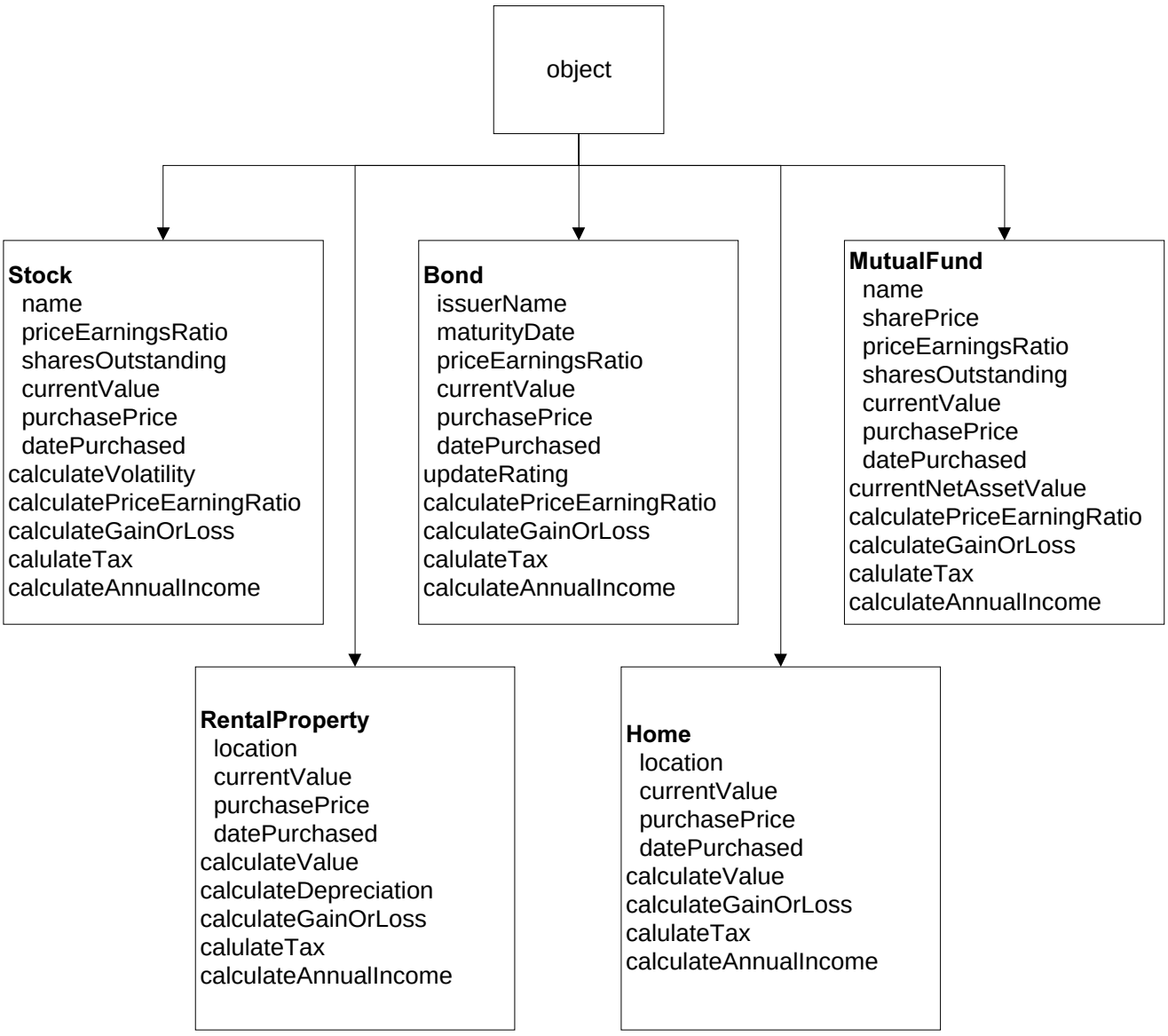
```
+ aNumber
        "Answer sum of the receiver and aNumber."
    | result |
    <primitive: 41>
    aNumber isFloat
        ifTrue: [
             result := self class basicNew: 8.
             FloatLibrary add: self to: aNumber result:
                    result.
            ^result]
        ifFalse: [^self + aNumber asFloat]
```

      - Integer:

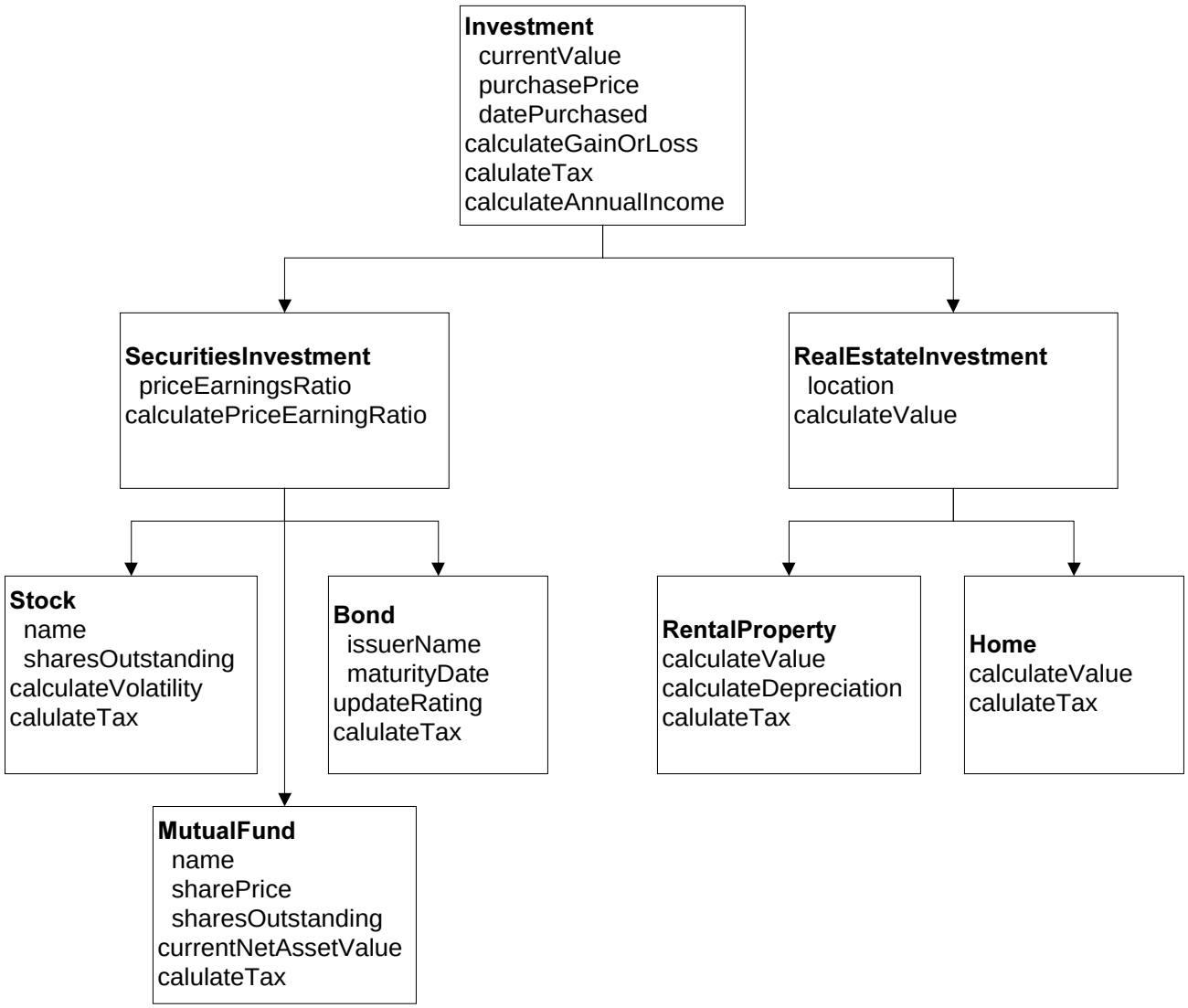
```
+ aNumber
        "Answer the sum of the receiver and aNumber."
    <primitive: 21>
    ^aNumber + self
```

Lecture 6: OO 4-Pass Process – an Investment Manager

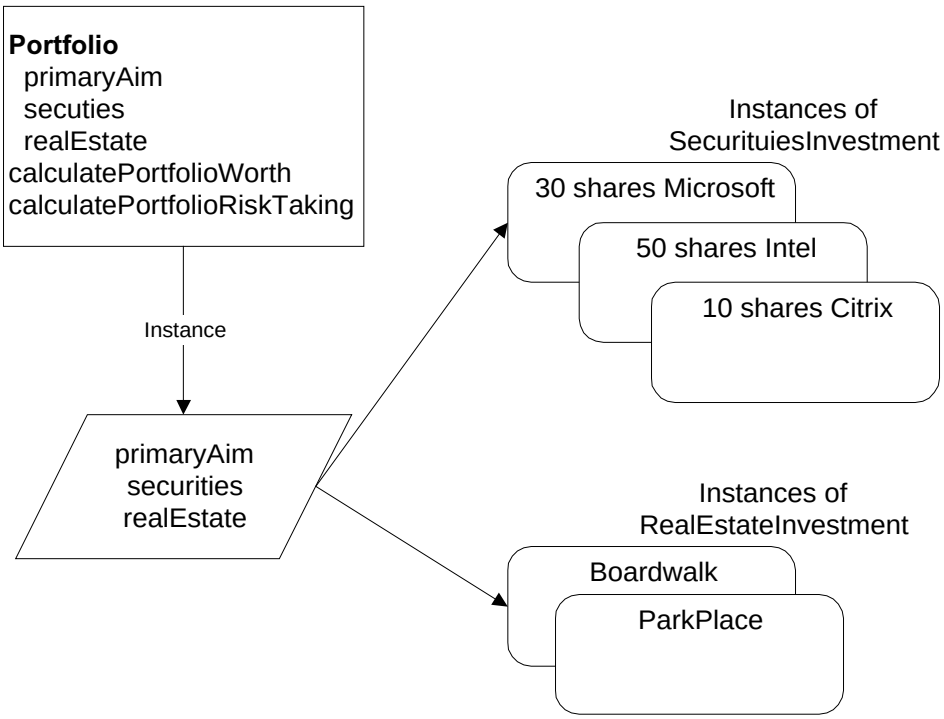
- **Pass 1: Abstraction**
  - Abstrction to share state/ behavior common to all investemnts
- **Pass 2: Abstraction**
  - Abstraction to share state / behavior for securities objects vr. Real estate investment objects
- **Pass 3: Composition**
  - Composition to create a portfolio of investments with a primary investment plan
- **Pass 4: Factorization**
  - Factorization to make explicit an anaysis of economic conditions related to investments
- Problem Statement: Design an Investment manager to handle stocks, bonds, mutual funds, houses and rental property
- Initial Design
  - What functionality do all investments share?
    - They all have currentValue, purchasePrice and datePurchased instance variables and calculateGainOrLoss, calculateTax and calculateAnnualIncome methods.
    - These variables and methods can be considered as the basis of creating a new, abstract superclass for the investments.



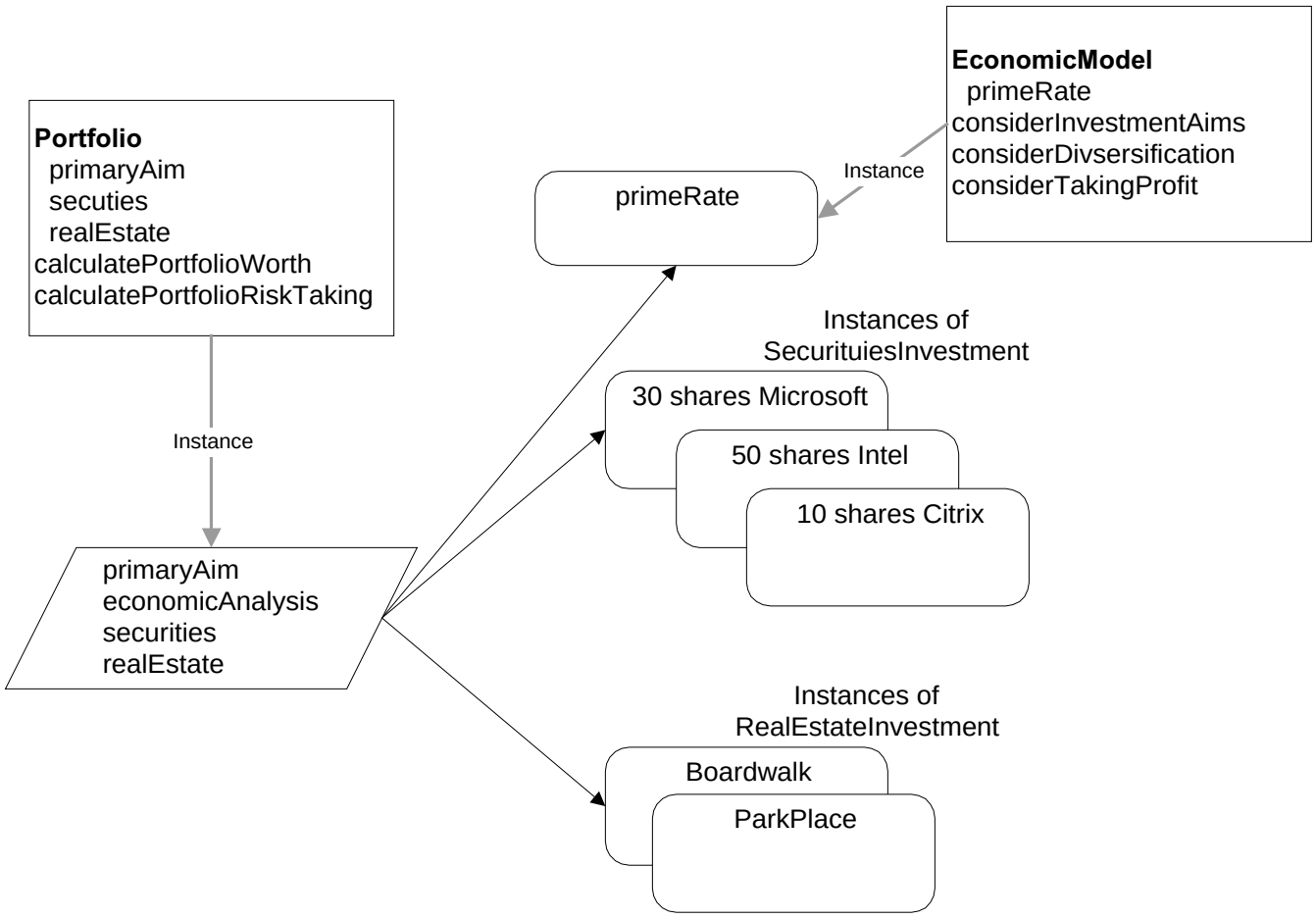
- Design Pass 1 (abstraction)
  - We can use abstraction to produce a new class, Investment. This is an abstact class that serves as the superclass for the concrete investment classes. It holds state variables and methods common to all investments



- Design Pass 2 (abstraction)
  - We now produce two new abstract classes:
    - SecuritiesInvestment to hold commonalties between Stock, Bond, and MutualFund.
    - RealEstateInvestemnt to hold commonalties between Home and RentalProperty.



- Design Pass 3 (composition)
  - Now we create a Portfolio class to hold all of the primary investment aim (risk level) and the collection of investments.
  - We'll create two state variables which hold the two collections of objects made up from the two classed defined in Pass 2.



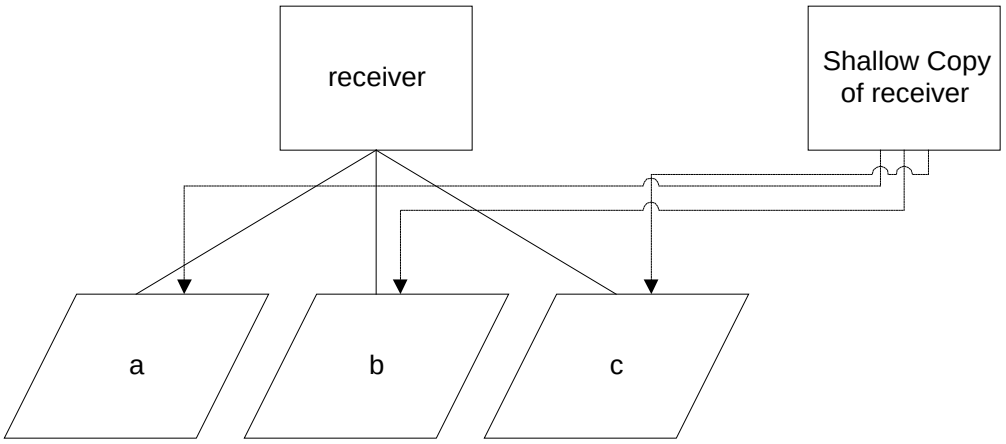
- Design Pass 4 (factorizarion)
  - In the final pass, we factor out “economic model” state and behavior as apotentially reusable part of Portfolio, and create the new class EconomicModel. This class lives outside the hierachy, and becomes part of the Portfolio via composition.
  - Remember factorization has two components
    - Break up large, complex classes into separate, more reusable components
    - Recover the original functionality through composition or inheritance.
  - How did we know to use composition instead of inheritance?
    - Which makes more sense:
      - “is-a” EconomicModel a Portfolio? (Inheritance)
      - Is an EconomicModel “part-of” a portfolio? (Composition)

Lecture 7: The Object Class

- The Object class is the main class from which all other classes are derived.
- Any and every kind of object in Smalltalk can respond to the messages defined by the Object class
- All methods of the Object class are inherited to overridden
- **Functionality of an object**
  - Determined by its class
  - Two ways to test functionality
    - Comparing object to a class or superclass to test membership or composition
      - `receiver isKindOf: aClass`
        - tests if the receiver is a member of the hierarchy of aClass
        - `anInteger isKindOf: Integer` returns true
      - `receiver isMemberOf: aClass`
        - tests if the receiver is of the same class
        - `anInteger isMemberOf: Magnitude` returns false
      - `receiver respondsTo: aSymbol`
        - tests if the receiver knows how to answer aSymbol
        - `anInteger respondsTo: #sin` returns true
        - `anInteger respondsTo: #at:` returns false
    - Querying the object for its class
      - `receiver class`
      - `#(1 2 3) class` returns Array
- **Comparison of objects**
  - Comparison and equivalence are very similar, but should not be confused
    - == is used to test if the receiver and argument are the same object
      - `#(a b c) class == Array` returns true
      - `#(a b c) == #(a b c) copy` returns false
    - = is used to test if the receiver and argument represent the same component
      - `#(a b c) class = Array` returns true
      - `#(a b c) = #(a b c) copy` returns true
    - Other comparison operations
      - `receiver ~= anObject`
        - Not equal
      - `receiver ~~ anObject`
        - Not Equivalent
      - `receiver hash`
        - `hash` provides a nice way of telling objects apart, too much trust should not be placed in comparing objects of the same class, as hash is often trivialized (as in the example below, `Array` uses size has the hash function).
        - Ex:

          ```
          a := 3.147 hash. ← 132
          b := 3.14 hash. ← 287
          c := #(1 2 3) ← 3
          d: = #(3 4 5) ← 3
          ```
      - `receiver hashMappedBy: map`
- **Copying objects**
  - `deepCopy` has been removed since VisualWorks 1.0
  - Two methods for copying:
    - `copy` returns another instance just like the receiver. Usually `copy` is simply a shallow copy, but some classes override it.
      - Does not copy the objects that the instance variables contain, but ciopies the "pointer" to the objects.

- `shallowCopy` returns a copy of the receiver which shares the receiver's instance variables. This allows two objects to share one set of instance variables.



- `deepCopy` must be implemented in the rare cases in which it is needed
  - How should this be done? Create new instances of the member objects, then assign them to the new object.
- Example, shallow copies of arrays.:

```
| array1 array2 object1 object2 object3|
object1 := #('A').
object2 := #('B').
object3 := #('C').
array1 := Array with: object1 with: object2 with: object3.
array2 := array1 copy.
(array1 at: 1) at: 1 put: 'D'.
array1 inspect.
array2 inspect.
```

- **Accessing indexed variables**
  - `at: index` returns the object at index
    - `#(a b c) at: 2` returns 'b'

- `at: index put: anObject` puts anObject at index of the receiver
  - returns anObject
  - `#(a b c) at: 4 put: #d` returns 'd'
- `basicAt: index` is the same as `at: index` but cannot be overridden
- `basicAt: index put: anObject` – Same as above
- `size` returns the number of index in the receiver
  - `#(a b c d) size` returns 4
- `basicSize` same as size, but cannot be overriden
- `readFromString: aString` creates an object based on the contents of aString
- `Yourself` returns the receiver

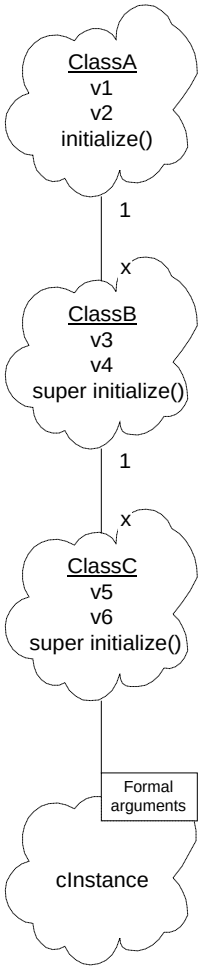
Lecture 8: Messages & Methods

- Messages are what is passed between objects
- Methods are what is defined in a class to act on an instance of the class
- **Message Expressions**
  - Receiver-object message-selector arguments
  - Unary
    - Receiver message-selector
    - Parsed left to right
    - Ex: `Time now.`
    - Ex: `8 squared.`
  - Binary
    - Receiver message argument
    - Parsed left to right
      - Ex: `1 + 2 * 3.` (Note: returns 9)
    - Parenthesis do the expected
      - Ex: `1 + (2 * 3).` (returns 7)
  - Keyword
    - Receiver message arguments
    - Ex:

      ```
      aString = 'ABC'.
      aString at: 3 put: $D.
      ```

      (Note: returns 'D', aString equals #(ABD))
      - Important to note that `'ABC' at: 3 put: $D` returns `$D`
      - `aString` is the object
      - `at` is the keyword message-selector
      - `3` is the argument
        - `'C'` is the object
        - `put` is the keyword message-selector
        - `$D` is the argument ($D is a literal)
  - Parentheses change order
  - Precedence *always* left to right
  - Separated by periods, unless temp variable declaration or comment
- **Method Lookup**
  - A method and a message-selector must be exactly the same, or no method will be found by the method lookup
  - The methods defined for the receiver's class first
  - If no match, the superclass is searched
  - Path continues through Object unless a method is found.
  - `self` refers to receiver, lookup starts within the class of the reciever
  - `super` refers to receiver, lookup starts in superclass of receiver

- Example
  - What is the order of initialization? (v1, v2, v3, v4, v5, v6)
  - Why? (initialize()'s look to superclass, then return to call their own initialize because they are implemented as `super initialize`)



Lecture 9: Variables and Return values

- A variable is a reference to any kind of object
- **Method arguments**
  - Accessibility: private
  - Scope: statements within the method
  - Extent: life of the method
  - Declaration: define with method name on first line of method (`name: aString`)
  - Assignment: Assigned by sender of the message (`aNode name: 'Node2'`)
  - Accessing : Directly by name
  - Ex: `anInteger raisedToInteger: 4.`
    - To understand this, it is easiest to look at literals and constants used as method arguments. The argument 4 is only visible to the object and the method- it cannot be accessed outside of the method. This coincides with the life of the variable, as it dies after the method call.
- **Temp variables**
  - Accessibility: private
  - Scope: statements within the method
  - Extent: life of the method
  - Declaration: use vertical bars
  - Assignment: use 'gets' operator
  - Accessing : Directly by name
  - Example:

```
cubeWithInteger
| x |
       x = self raisedToInteger: 3.
```

    - x is created in the method using the vertical bars, and is released once the method is finished.
- **Instance variables**
  - Accessibility: private
  - Scope: Instance methods of the defining class & subclasses
  - Extent: life of the instance
  - Declaration: define on the instance side of the class template

```
Object subclass #Node
       InstanceVariableNames: 'name nextNode'
       ClassVariableName: ''
       PoolDictionaries:''
       Category: ''
```

  - Assignment: write a method that sets the value
  - Accessing : write a method that gets the value
    - Can be either named or keyed
      - If keyed, then they can be accessed through ordinary `at:put:` messages
- **Class instance variables**
  - Accessibility: private
  - Scope: Class methods of the defining class & subclasses
  - Extent: life of the defining class
  - Declaration: define on the class side of the class template

```
Account class
       InstanceVaraibleNames: 'interestRate'
```

    - Assignment: write a class **initialize** method in the defining class and all of its subclasses
    - Accessing : Write a class method that returns the value
- **Class Variables**
    - Accessibility: shared
    - Scope: Instance and class methods of the defining class & subclasses
    - Extent: life of the defining class
    - Declaration: define on the instance side of the class template
    - Assignment: write a class **initialize** method
    - Accessing : Write a class method that returns the value
    - Always begin with uppercase
- **Global Variables**
    - Accessibility: shared
    - Scope: all objects, all methods
    - Extent: while in Smalltalk dictionary
    - Declaration: with assignment
    - Assignment: with declaration

```
Smalltalk at: #MyTranscript put: TextCollector new.
```

    - Accessing : Directly by name
    - Don't use, unless absolutely necessary. Bloated images, anti-OO code, incorrect code are the consequences.
- **Return Values**
    - Method always returns an object
        - Default return value is `self`.
        - Use ^ to explicitly return a different object
        - Can use both implicit and explicit returns in a method (i.e. in a conditional)

Lecture 10: Blocks and Branching

- **Blocks**
  - Contains a deferred sequence of expressions
  - Used in many of the control structures
  - Instance of `BlockClosure`
  - Returns the result of the last expression (similar to lisp)
    - Ex: `[3+4. 5*5. 20-10]` returns a Home Context with value of 10.
      - `[3+4. 5*5, 20-10] value` returns 10.
    - Ex: `['Visual','Works'] value` returns 'VisualWorks' (comma is binary method)
  - Syntax

    ```
    [:arg1 :arg2 … :arg255 | |temp vars| executable expressions]
    ```
  - A block can contain:
    - 0 to 255 arguments
    - temp variables
    - executable expressions
  - Block with no arguments: sequence of actions takes place every time value message is received by the block
  - Block with arguments: action takes place every time block receives messages value, value: value, etc.
  - block variables scope is only within defining block
  - NOTE: temp variables inside declared blocks have not been successfully tested with Smalltalk Express or GNU Smalltalk.
  - Examples
    - `[ :x :y | x + y / 2 ] value: 10 value: 20` (returns 15)
    - `[ |x| x := Date today. x day ]` value (returns the day to today's Date)
      - `[Date today day ] value` returns same value & is more succinct
    - `[ :y | |x| x := y *2. x * x ] value: 5` (returns 100)
    - `#(5 10 15) collect: [ :x | x squared]` (returns #(25 100 255)
      - sends 1 argument 3 times and collects the results into an array
- **Class Boolean**
  - Classes `True` and `False` are subclasses of Boolean
  - Logical operators can be used for testing
    - The 'and' operator: `&`
    - The 'or' operator: `|`
    - The negation operator: `not`
      - `not` is a unary operator
    - The equivalence operator: `eqv`
    - The exclusive or operator: `xor`
  - The `Boolean` classes are used in branching
    - `and`: and `or`: methods used with alternative blocks returns values of alternative blocks
    - `ifTrue:` and `ifFalse` are used with blocks to provide if-then support
      - can be used together in either order, or separately
- **Branching (Control Structures)**
  - Boolean classes `True` and `False` understand keyword messages:
    - `ifTrue:`
    - Ex: (result: anArray = #('a' 'b' 'c'))

```
| anArray |
anArray := #('a' 'b' 'd').
(anArray at: 3) asString > 'c'
   ifTrue: [anArray at: 3 put 'c'].
```

- `ifFalse:`
- `ifTrue: ifFalse:`
- Ex: (result: upperArray = #('A' 'B' 'D'))

```
| anArray upperArray |
anArray := #('a' 'B' 'd').
upperArray := Array new.
upperArray := anArray collect:
    [:aString | aString asUpperCase = aString
           ifTrue: [aString]
           ifFalse: [aString asUpperCase]].
```

- `ifFalse: ifTrue:`
- These messages demand zero argument blocks as their arguments
  - Ex:

```
abs
       ^self < 0
              ifTrue: [0 – self]
              ifFalse: [self]
```

  - What happens here?
    - `self` is compared to 0
    - corresponding block is executed
    - `(-self)` or `self` is returned depending on which block was executed
- Repetition
  - `timesRepeat:` message
    - Ex: `5 timesRepeat [ Transcript show: 'This is a test'; cr ]`
  - `to:` message (similar to for loop)
    - Ex: `1 to: 15 by: 3 do: [:item | Transcript show: item printString; cr]`
- Conditional Iteration
  - Blocks can be used as arguments in messages and can be receiver objects
  - `whileTrue:` and `whileFalse:` messages
    - get sent to blocks. `ifTrue:` and `ifFalse:` get sent to Boolean
    - Ex (receiver):

```
Initialize: myArray
| index |
index := 1.
[ index <= myArray size]
       whileTrue:
       [myArray at: index put: 0.
       index := index + 1]
```

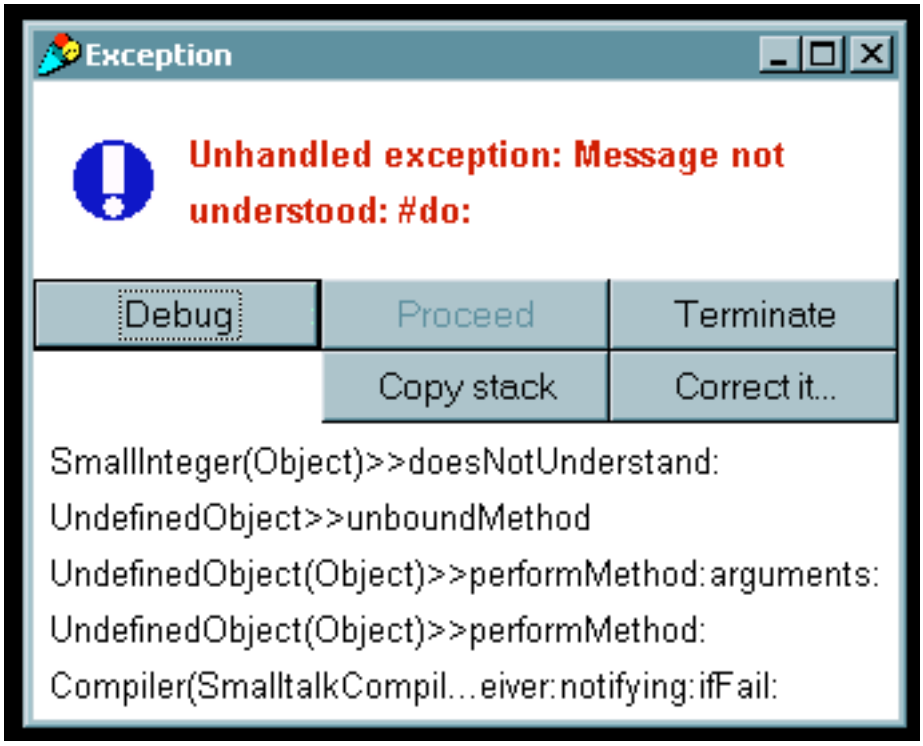
    - Ex (argument):

```
Initialize: myArray
| index |
index := 1.
[ myArray at: index put: 0.
       index := index + 1.
       index  <= myArray size] whileTrue;
```

Lecture 11: Reporting Errors and Debugging techniques

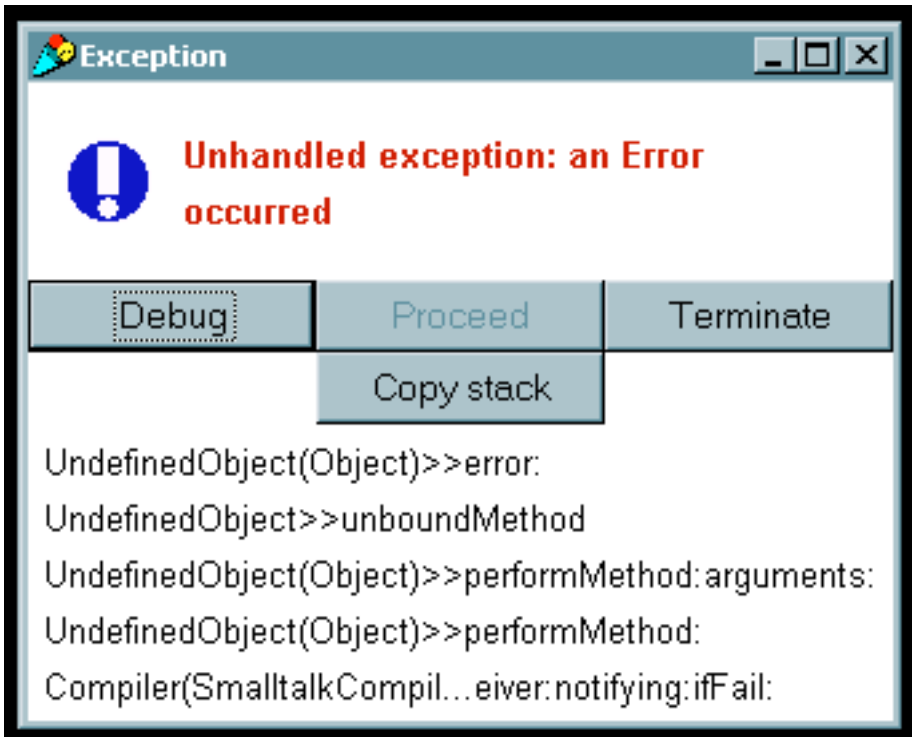
- **Error Handling**
  - Smalltalk's interpreter handles all errors
  - An error is reported by an object sending the interpreter the message `doesNotUnderstand: aMessage`
  - There are some common error messages supported in the Object class, but implementation is dependant on the system
    - `doesNotUnderstand: aMessage`
      - Lets look at an example of trying to use a method that an object of the class SmallInteger cannot understand.

```
| anInteger |
anInteger := 0.
self doesNotUnderstand: (anInteger do:[]).
```

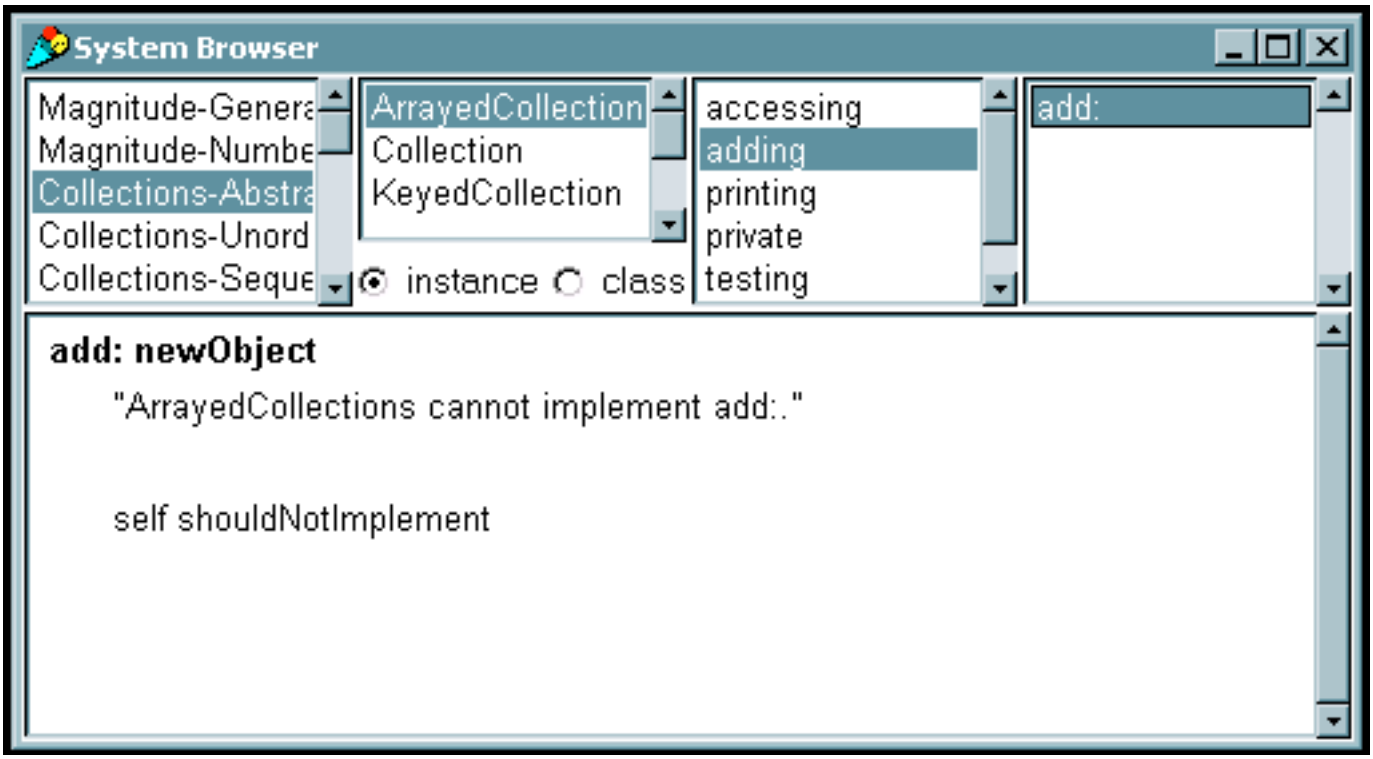


- `error: aString` uses aString in the report the user sees

```
self error: 'an Error occurred'.
```



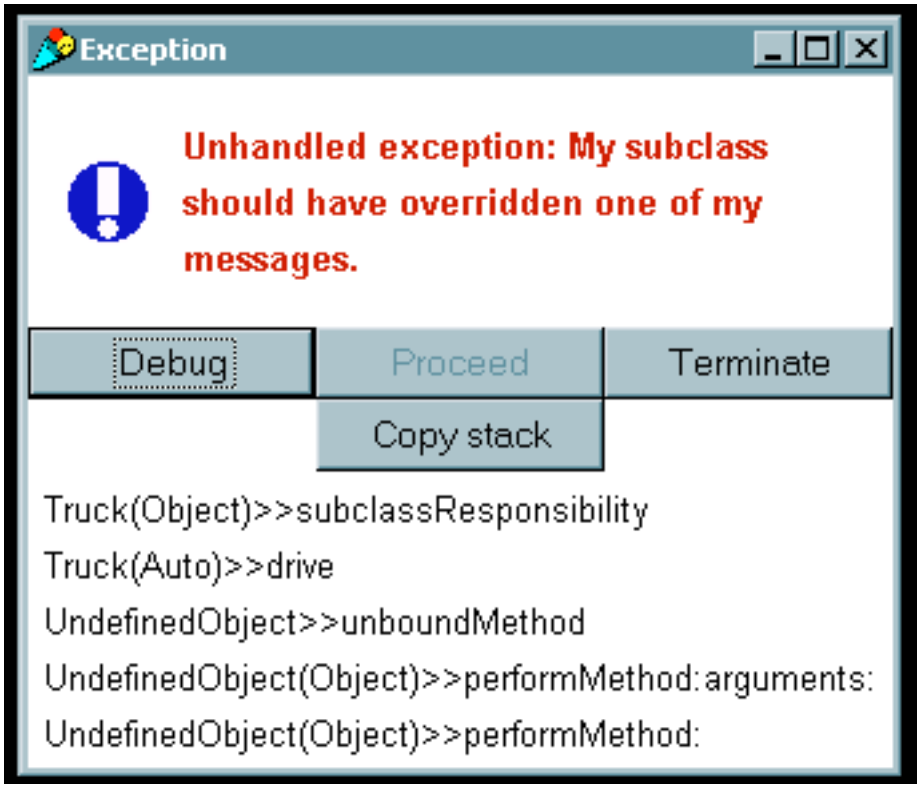
- `primitiveFailed` reports that a method implementing a system primitive failed
- `shouldNotImplement` reports that the superclass says a method should be implemented in the subclasses, the subclasses do not handle it correctly.
  - This method is utilized throughout the collection classes. If we look at the Array class, we'll see this method is used inside the add: method.
    - Arrays are statically sized collections, and the add: method is used to grow the size of collections.



- `subclassResponsibility` reports that a subclass should have implemented the method
  - This method is used extensively in abstract classes. This method allows all objects in the hierarchy to implement a method differently, while reporting an error if the method was not defined.
    - Example: Class Auto defines a method `drive`, but only calls the `subclassResponsibility` method. We define a subclass Truck, but do not define the method `drive`. If we then define a Truck object and call the drive method, then Smalltalk will try to pass the drive message up the tree until a parent class knows how to implement it- in this case displaying a `subclassResponsibility` error message.

```
Object subclass: #Auto
   instanceVariableNames: 'speed '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples-General'!


!Auto methodsFor: 'creation'!

withSpeed: aSpeed

   self subclassResponsibility! !

!Auto methodsFor: 'driving'!

accelerate

   speed := speed + 1.!

decelerate
```

```
    speed := speed + 1.!

drive

    self subclassResponsibility! !

Auto subclass: #Truck
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Examples-General'!


!Truck methodsFor: 'creation'!

withSpeed: aSpeed

    speed := aSpeed.! !


| aTruck |
aTruck := (Truck new) withSpeed: 5.
aTruck drive.
```



- **Message Handling**
  - Used to send messages to objects, usually only created when an error occurs
  - `perform:` is the method called to pass messages, takes many different arguments, or just aSymbol.
    - A good example of this can be seen in the Goldberg book (page 245).
      - Suppose we wish to write a simple calculator that checks to make sure each operator is a valid operator.

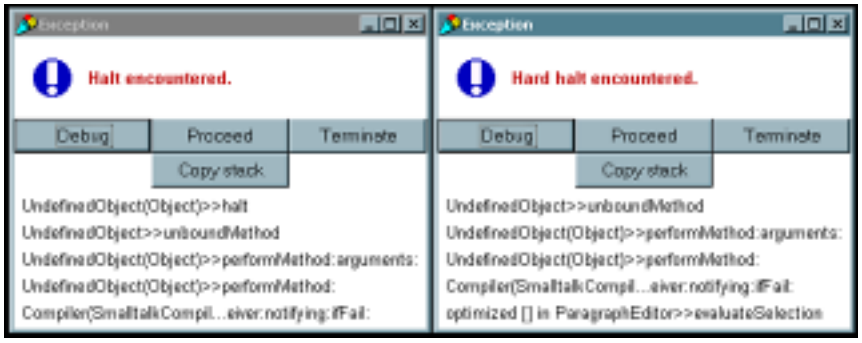
```
Object subclass: Calculator
   instanceVariableNames: 'result operand'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples-General'!

!Calculator methodsFor: 'creation'!

new
   ^super new initialize


!Calculator methodsFor: 'accessing'!

result
   ^result

!Calculator methodsFor: 'calculating'!

apply: operator
   (result respondsTo: operator )
          ifFalse: [self error: 'operation not understood'].
   operand isNil
          ifTrue: [result := result perform: operator]
          ifFalse:
               [result := result perform: operator with: operand]

clear
   operand isNil
          ifTrue: [result := 0]
          ifFalse: [operand := nil]

operand: aNumber
   operand := aNumber

!Calculator methodsFor: 'private'!

initialize
   result := 0
```

- The following code shows an example of how to use the class Calculator

```
| aCalculator |
aCalculator := Calculator new. "result = 0"
aCalculator operand: 3.
aCalculator apply: #+. "result = result + 3 ← 3"
aCalculator apply: #squared. "result = 3 ^ 2 ← 9"
aCalculator operand: 4.
aCalculator apply: #-. "result = result – 4 ← 5"
```

- System Primitive Messages
  - Messages in class Object used to support system implementation
  - `InstVarAt: anInteger` and `instVarAt: anInteger put: anObject` are examples which are used to retrieve and store instance variables.
  - In general, these will not be used, but are important to how Smalltalk works.
- **Class UndefinedObject**
  - the object `nil` represents a value for uninitialized variables
  - `nil` also represents meaningless results
  - Testing an object's initialization is done through `isNil` and `notNil` messages

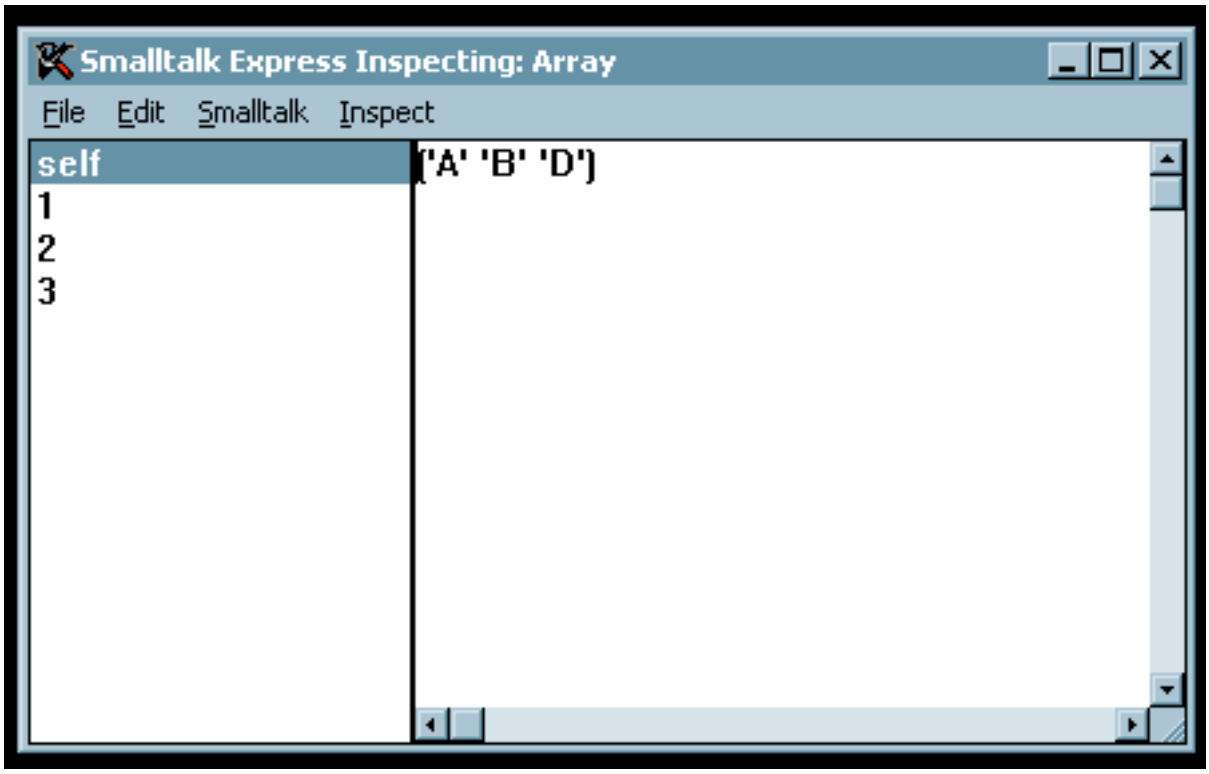
- **Debugging**
  - Smalltalk has a small set of methods for error handling and are useful to debugging. These messages are implemented by passing Signals.
    - Whats a signal? A signal is an Exception passed to the VM. A signal will stop the execution and show a window with a message and has several qualities, such as whether or not the exception is proceedable. An example of this is the `halt: aString` message, which raises a `haltSignal` with the context of the receiver and the error message of `aString`.
      - `errorSignal`
      - `messageNotUnderstoodSignal`
      - `haltSignal`
      - `subclassResponsibilitySignal`
  - `confirm`: similar to `notify:` method, brings up a window asking for confirmation, not in all implementations . In VW 3.0 and above, the `confirm:` method belongs to class Dialog.
    - Ex: `(Dialog confirm: 'Quit ?') ifTrue:[aBlock].`
  - **`halt`**
    - `halt` shows the debug window, with 'halt encountered' or similar message as the primary error. Useful for setting a breakpoint to check value of variables
      - Ex: `self halt.`
      - Ex: It is possible to stop other objects
      `Transcript halt.`
    - `halt: aString` implements halt, bringing up a window with the label from `aString`
    - `halt`: appears very similar to notify:, but with one difference. `halt`: allows invariants related to multiple processes to be restored.
      - How could I do this? When execution halts, use the workspace to restore the values.
    - `hardHalt` halts the execution without passing a signal.



    - `notify: aString`: shows a message dialog window with aString as the label. This method is not available in Smalltalk Express.
      - Ex: `self notify: 'custom error message'.`
    - `inspect` displays a window showing the object and all of its variables
    - Ex:

```
| anArray upperArray |
anArray := #('a' 'B' 'd').
upperArray := Array new.
upperArray := anArray collect:
      [:aString | aString asUpperCase = aString
            ifTrue: [aString]
```

```
            ifFalse: [aString asUpperCase]].
upperArray inspect.
```



Lecture 12: Designing and implementing classes

- **Steps to develop a specification**
  1. Decide what we want the program to do
  2. Decide on the data structures
  3. Decide on the operations we want to apply to these data structures
- **The message protocol**
  - Class Protocol: A description of the protocol understood by a class
  - Typically contains protocols for creating and initializing new instances of the class
  - Instance Protocol: A description of the protocol understood by instances of a class
  - Messages that may be sent to any instance of the class
  - **Steps to implementing a class**
    1. Deciding on a suitable representation for instances of the class.
    2. Selecting and implementing efficient algorithms for the methods or operations
    3. Deciding on class variable and instance variables
- **Describing a class**
  - Class name: A name that can be used to reference the class
  - Superclass name: name of the superclass
  - Class variables: variables shared by all instances
  - Instance variables: variables found in all instances
  - Pool dictionaries: Names of lists of shared variables that are to be accessible to the class and its instances. Can also be referenced by other unrelated classes
  - Class methods: operations understood by the class
  - Instance methods: operations that are understood by instances
  - Example: A class for complex numbers
    - Step 1: What do we want to be able to do?
      - Specify real and complex parts
      - Do simple operations of complex and real parts
    - Step 2: What do we want to use?
      - Specify real and complex parts
    - Step 3: How are we going to use the data structures?
      - Creating a complex number
      - Accessing complex and real parts
      - Adding and Multiplying Complex numbers
    - The Class Description (for more detail refer to LaLonde pages 44-45)

```
Class Complex
Class name                Complex
Superclass name                Object
Instance variable names   realPart imaginaryPart

Class methods

Instance creation

newWithReal: realValue andImaginary: imaginaryValue
       "Returns an initialized instance"
       | aComplex |
       aComplex := Complex new.
       aComplex realPart: realValue;
              imaginaryPart: imaginaryValue.
              ^aComplex


accessing
```

```
realPart
       “Returns the real component of the reciever”
       ^realPart

imaginaryPart
       “Returns Imaginary part”
       ^imaginaryPart

operations

+ aComplex
       “Returns the receiver + aComplex”
       | realPartSum imaginaryPartSum |
       realPartSum := realPart + aComplex realPart.
       imaginaryPartSum := imaginaryPart + aComplex imaginaryPart.
       ^ Complex newWithReal: realPartSum andImaginary:
              imaginaryPartSum.

* aComplex
       “Returns the receiver * aComplex”
       | realPartProduct imaginaryPartProduct |
       realPartProduct := (realPart * aComplex realPart) –
                     (imaginaryPart * aComplex imaginaryPart).
       ComplexPartProduct := (realPart * aComplex imaginaryPart) +
                     (imaginaryPart * aComplex realPart).
       ^ Complex newWithReal: realPartProduct andImaginary:
              imaginaryPartProduct.
```

- The following code shows how to use this new class. The code computes the magnitude of the complex number. After multiplying the number by its conjugate, there is only a real part, so we just take the square root.

```
| aNumber |
aNumber := (Complex new) newWithReal: 1 andImaginary: 1.
aNumber := aNumber * (Complex new)
       newWithReal: (aNumber realPart)
       andImaginary: (0 – aNumber imaginaryPart).
(aNumber realPart) sqrt.
```

Lecture 13: VisualWorks

Note: The lectures on VisualWorks were taken from http://www.cs.clemson.edu/~lab428/VW/VWCover.html. Only minor modifications have been made.

Starting VisualWorks

To start VisualWorks from the command line of a Unix system, use the command

/usr/local/visual/vw image.im

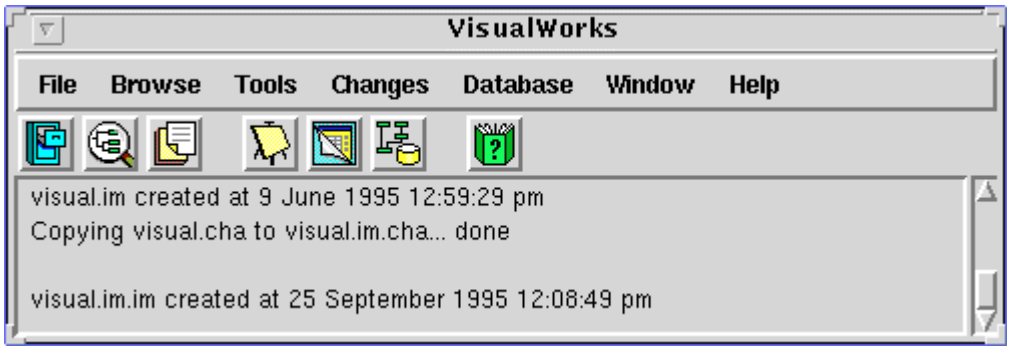
To start VisualWorks with an image other than the default, use the command

vw *image-file*

Enter the appropriate command to start VisualWorks on your system. You should see two windows, the VisualWorks Launcher and the Workspace.

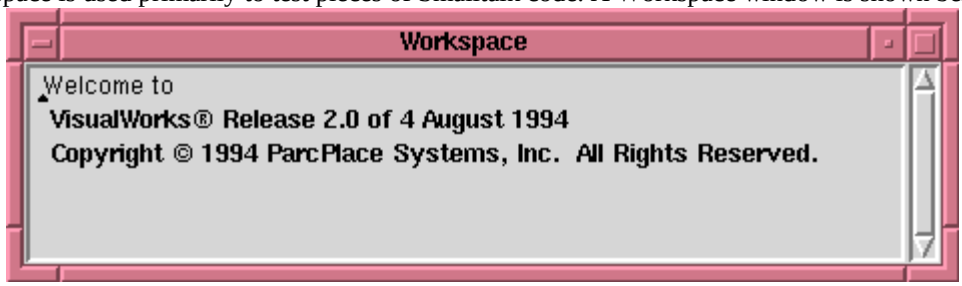
VisualWorks Launcher

The VisualWorks Launcher is the main window in VisualWorks. It is used primarily to access the various tools and resources available. A Launcher window is shown below.



Workspace

A Workspace is used primarily to test pieces of Smalltalk code. A Workspace window is shown below.



Using the Mouse and the Pop-Up Menus

General familiarity with windowing systems is assumed in this tutorial. Mouse button operations refer to the left ([Select]) mouse button unless otherwise specified.

There are two types of pop-up menus associated with each window in VisualWorks. There is the [Window] menu which is accessed by clicking the right mouse button in the window. The [Window] menu is used for closing, moving, and resizing the current window. The second pop-up menu is the [Operate] menu which is accessed by clicking the middle mouse button in the window. There may be more than one [Operate] menu per window, in which case an area will be specified in which to click the middle mouse button. To select an item from either the [Operate] or [Window] menus the mouse button used to obtain access the menu must be used.

Note: The following conventions are used for one-button and two-button mice:

Two-button mouse

The left button is the [Select] button. The right button is the [Operate] button. The [Window] menu is obtained by using the Control key and right ([Operate]) button together.

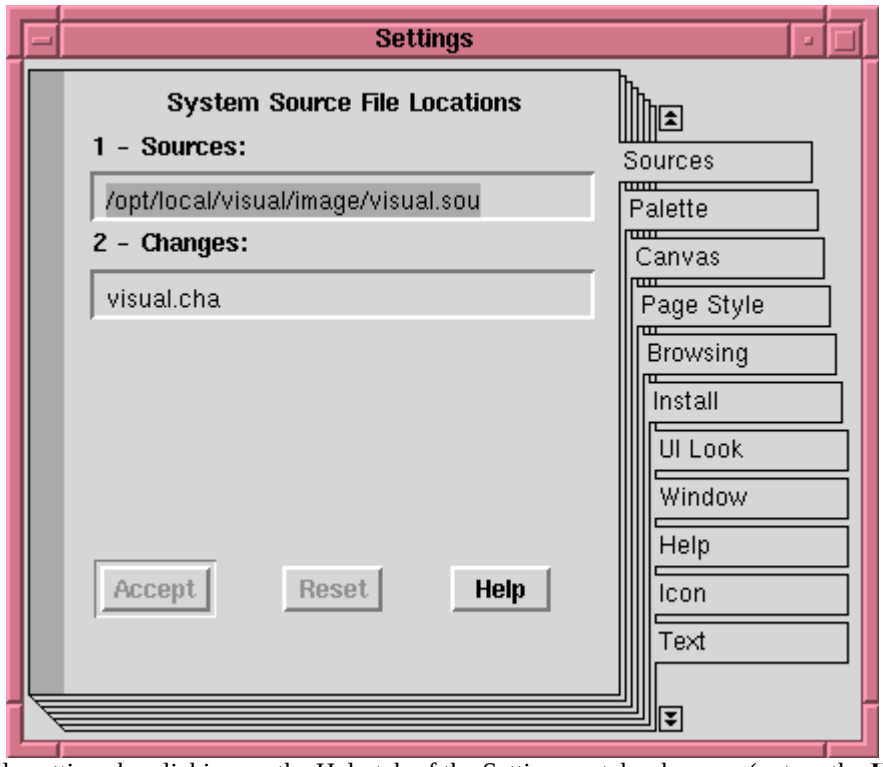
One-button mouse

The button alone is the [Select] button. The [Operate] menu is accessed using the Option key and the button together. The [Window] menu is accessed using the Command key and the button together.

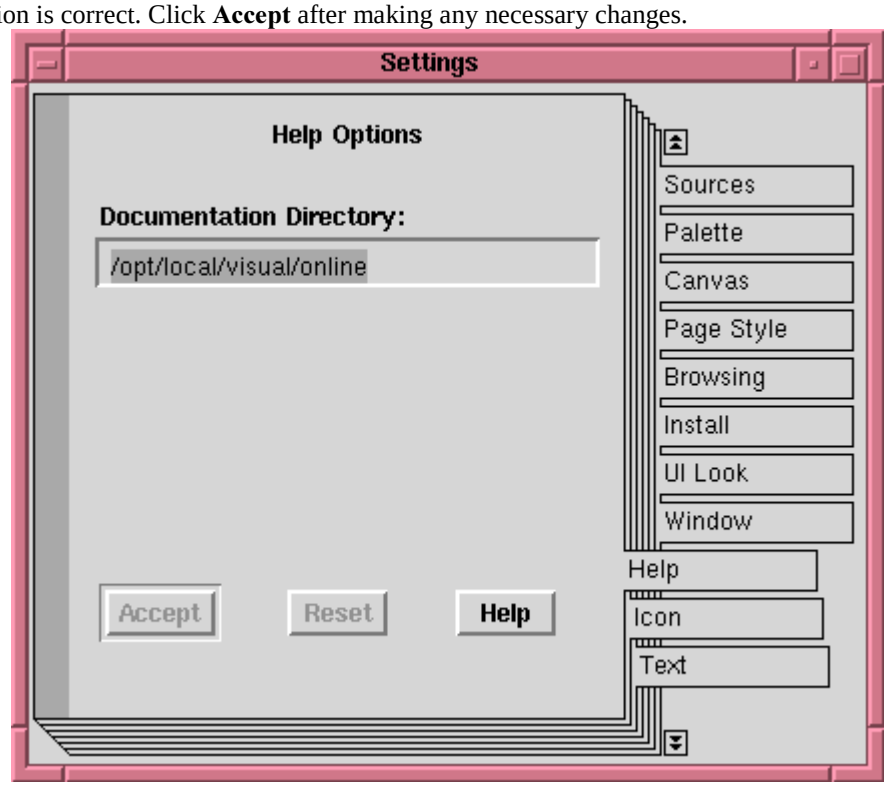
In windows that have a menu bar, pulldown menus are accessed by clicking on the word associated with the menu. For example, click on **File** located on the Launcher's menu bar to obtain the File pulldown menu. Pulldown menu selections will be specified by the menu name followed by an arrow(-) and the menu item. For example, the **File-Settings** option from the VisualWorks Launcher refers to the Settings option from the File pulldown menu. Many pulldown menu options also have a shortcut button on the tool bar, which will be refer to with its associated icon. For example, the Canvas Tool may be obtained by using either the **Tools-New Canvas** menu option or by the shortcut button  .

Setting up VisualWorks

To insure access to Smalltalk source code and VisualWorks On-line Documentation, the proper paths must be set using the Settings window. Open the Settings window by selecting the **File-Settings** options from the VisualWorks launcher. You should see the window depicted below. Make sure that the correct the path for the VisualWorks source code is displayed (*visual_path*/image/visual.sou). If you need to correct the path, correct as necessary and click **Accept**. (Note: No changes should be needed at Clemson.)



Select the help settings by clicking on the Help tab of the Settings notebook pages (not on the **Help** button for the Settings window). You should see the following window. Make sure that the path for the online documentation is correct. Click **Accept** after making any necessary changes.

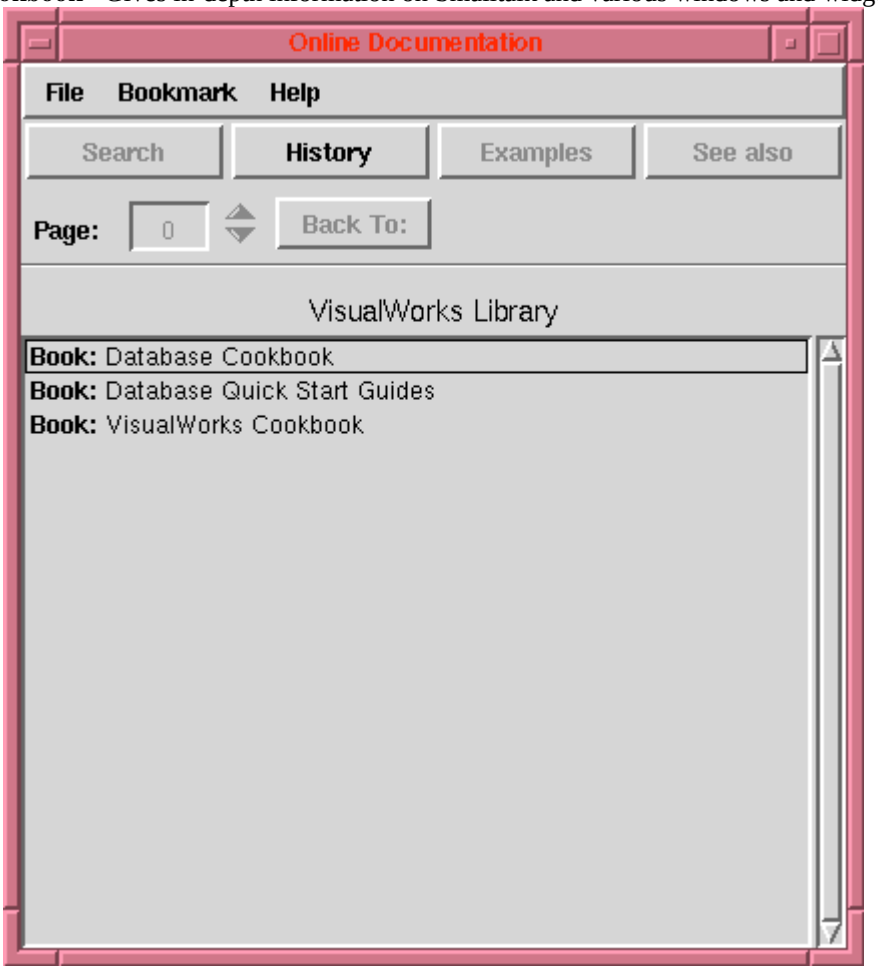


Now close the Settings window by selecting **close** from the Settings [Operate] menu.

Online Documentation

Another useful tool in VisualWorks is the online documentation. The online documentation can be accessed from the VisualWorks Launcher via the **Help-Open Online Documentation** option or the shortcut button  . Shown below is the Online Documentation window that lists three manuals that may be used as further references. These three manuals include the following:

Database Cookbook - Gives information on how to connect to a database.
Database Quick Start Guides - Gives information on how to create models for database applications.
VisualWorks Cookbook - Gives in-depth information on Smalltalk and various windows and widgets.



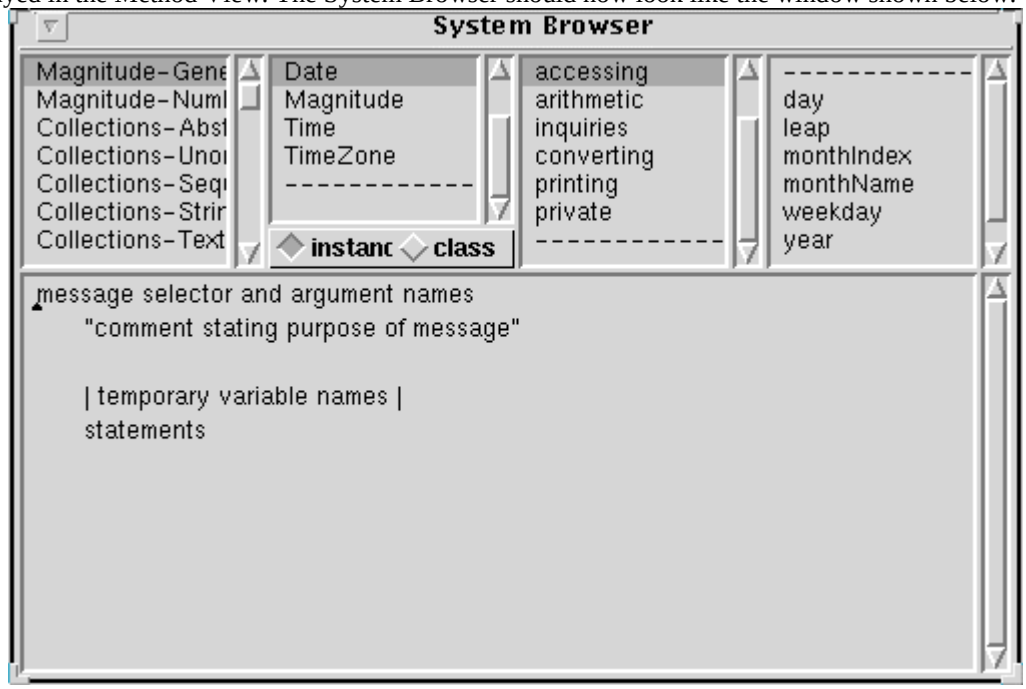
For example, suppose you needed information on how to construct a Smalltalk message. Select the **VisualWorks CookBook** by clicking on the book title with the mouse button. Now, select **Chapter 1: Smalltalk Basics**, and then select **Constructing a Message**. Information on your topic is now displayed in the Online Documentation window. Close the Online Documentation window.

System Browser

A System Browser is a useful tool for viewing Smalltalk classes, protocols, and methods. Not only does a Browser provide useful ways to view system and user classes, it also has many features that help the user to quickly and easily develop classes, protocols, and methods.

To open a System Browser, select **Browse-All Classes** from the VisualWorks Launcher or use the shortcut button  . Notice that a System Browser is divided into four columns across the top half of the window, and the bottom half contains a text area. These are important areas to learn. The columns (left-to-right) are the Category View, the Class View, the Protocol View, and the Method View. The text area that comprises

the bottom half of the window is the Code View. These five different views will be referred to frequently in the development portion of this tutorial.
For example, select the category "Magnitude-General" and the classes associated with that category appear in the Class View. Select the Date class, and the protocols associated with that class are displayed in the Protocol View. Finally, select the accessing protocol and the methods associated with that protocol are displayed in the Method View. The System Browser should now look like the window shown below.

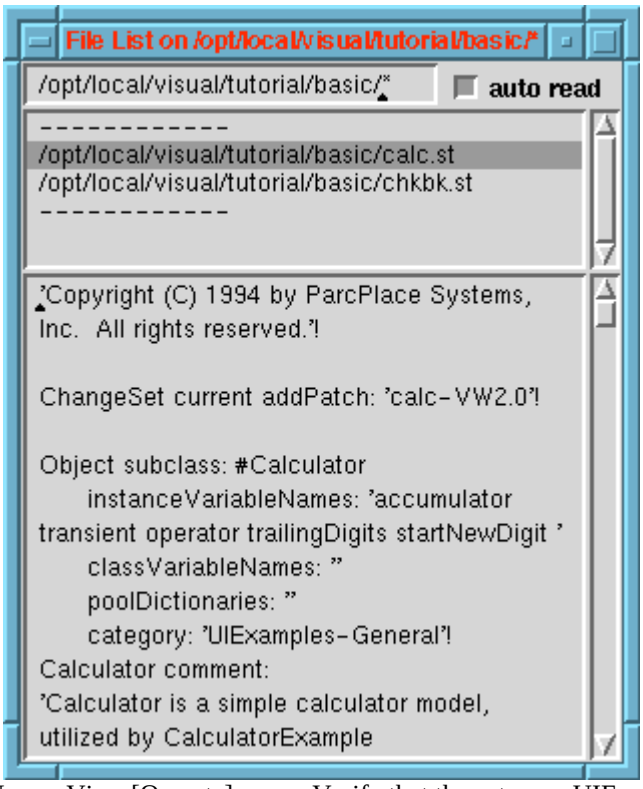


Notice that the Code View currently contains only a template for the code of a method. Select any method and you will see its code in the Code View. Close the System Browser by selecting **close** from the System Browser [Window] menu.

Filing In and Filing Out Components
To save categories, classes, or even methods you can write ("file out") these components to a file and then remove them from your image. Later you can read ("file in") these components into your image.

Filing In

We will illustrate how to "file in" components by adding an application, Calculator Example, to our image. The CalculatorExample class is in the category UIExamples-General, and it is stored in the file *visual_path*/tutorial/basic/calc.st. First note that the category UIExamples-General is not currently in the image by scrolling through the categories in the Category View of a System Browser. Open a File List from the **Tools-File List** option or the shortcut button  of the VisualWorks Launcher. Enter /opt/local/visual/tutorial/basic/* in the first input field, which is called the Pattern View, and Return. (Note: This is for the *visual_path* at Clemson.) Select **/opt/local/visual/tutorial/basic/calc.st** from the file list, which is called the Names View. The File List should look like the following window.



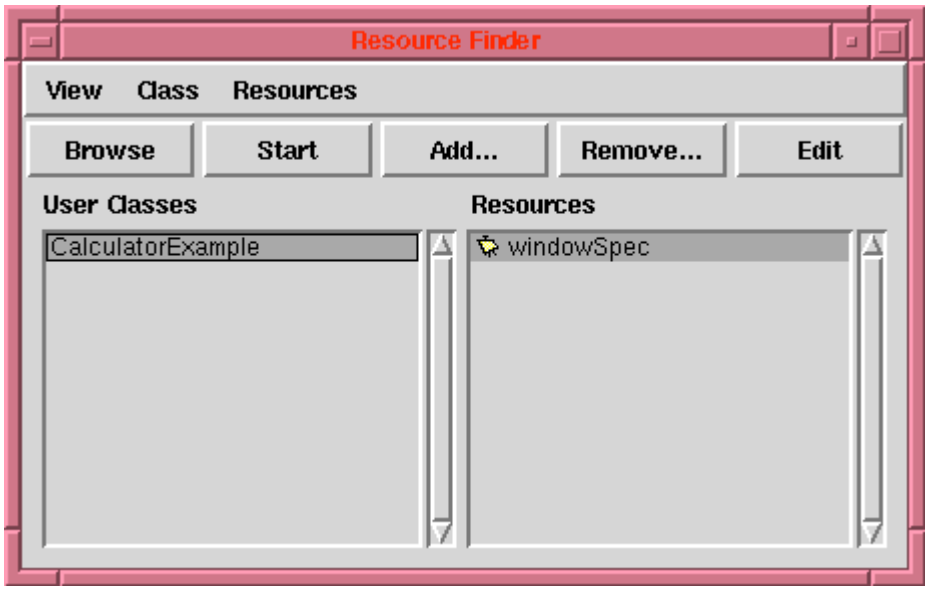
Select **file in** from the Names View [Operate] menu. Verify that the category UIExamples-General is now in the image by using the System Browser. Close the File List.

Filing Out

You can file out a category, class, or even single methods. For example to file out a category, select (with a mouse click) a category from the System Browser (so that the category is highlighted). Select **file out as...** from the Category View [Operate] menu, enter the file name to which you wish to file the category out, and click **OK**. A category, class, or method that is filed out can later (for example, in another VisualWorks session) be filed in as illustrated in the previous section.

Starting an Application

Once you have developed an application you will want to execute it. To start a completed application, open a Resource Finder using **Browse-Resources** from the VisualWorks Launcher or the shortcut button  . Select **View-User Classes** from the Resource Finder menu. Select the class you would like to start. To start the Calculator Example that we previously filed in, select the **CalculatorExample** class and the **windowSpec** resource as depicted below. (Note that the windowSpec resource is automatically selected because it is the only resource for the CalculatorExample class.) Select **Start** from the Resource Finder and the Calculator Example will start. When you have finished using the Calculator, close the application by selecting **close** from the Calculator [Window] menu. Close the Resource Finder.



A class may have one or more "resources", which are user interfaces. To start an application, we select its class and the appropriate resource for the initial window of the application.

Saving Your Work

Doing a "Save" in VisualWorks is a complete save. It actually saves an image of all of the current classes (system and user), active windows, etc. This is a nice feature if it becomes necessary to stop in the middle of your work. Unfortunately, saving your image has drawbacks. An image on a Solaris platform will take up approximately 4 megabytes of disk space. To save an image, select **File-Save As** from the VisualWorks Launcher. A dialog box will appear. Enter the name for your image file and click OK. VisualWorks will save the file in the current directory unless a different path is specified. The file will be have the extension .im.

VisualWorks automatically creates a .cha file in the directory from which VisualWorks is started, and VisualWorks periodically records the changes made to the initial image in the .cha file. The .cha files can be useful for change management, and they can sometimes be used for error recovery (e.g., if you mistakenly delete some work that you need or fail to file out some work that you wished to save), but you may wish to delete the .cha files until you use VisualWorks in a large project.

Lecture 14: More on the Basic VisualWorks Environment

The purpose of this Lecture is to provide a further introduction to the basic VisualWorks environment for the support of Smalltalk.

VisualWorks includes many tools that facilitate the development of Smalltalk programs. These tools were introduced in Chapter 2, and this chapter provides further illustrations of the uses of the tools for implementing Smalltalk programs. The use of VisualWorks for developing GUI applications will be illustrated in Chapters 4-6.

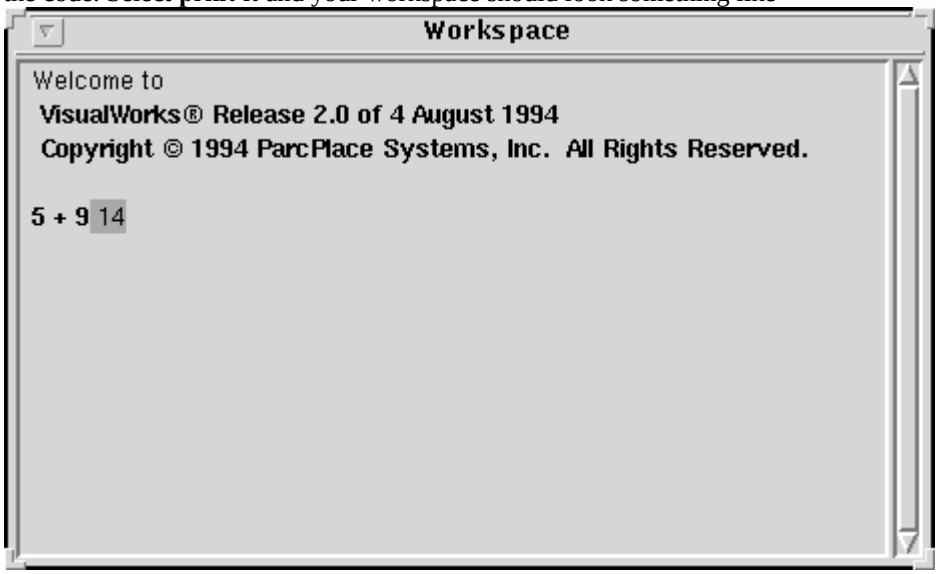
Workspaces

If you do not currently have VisualWorks started, you should start it now. VisualWorks initially displays a Launcher and a Workspace. The Launcher contains control widgets for various VisualWorks facilities, as discussed in Chapter 2, and it also includes a Transcript window in the lower part of the window. We will illustrate some of the facilities of VisualWorks using the Workspace for Smalltalk and the Transcript for displaying results. You should resize these windows if needed so that they are large enough for several lines of text.

You can type segments of Smalltalk code into a workspace (or most any other VisualWorks window, for that matter) and execute it. For example, type

```
5 + 9
```

in the workspace. (You should move the cursor down to a new line with the mouse select button and/or the arrow and return keys first.) Now highlight 5 + 9 by dragging the mouse [Select] across the text. From the [Operate] (middle button) menu, note that you can **do it** or **print it**. Selecting **do it** will cause the code to be executed, and selecting **print it** will cause the code to be executed and the result printed immediately following the code. Select **print it** and your workspace should look something like



(Selecting **do it** here will have no visible effect, because evaluating 5 + 9 does not have any external effect (side effect).) Note that the result printed is highlighted, so it can easily be deleted by pressing the Backspace key.

Testing code in this way is useful for code development in Smalltalk and also for debugging. Remember that you can highlight Smalltalk code in most any window and execute it or print its result in this manner. Multiple statements, separated by periods in the usual way, can be executed with a single **do it** (or **print it**).

The Transcript
The transcript window in the lower part of the Launcher is associated with the Smalltalk global variable "Transcript". Transcript is an instance of the class TextCollector that allows text to be displayed in the transcript window. Strings can be displayed in the transcript window by sending a show: message with a string argument to Transcript. For example,

```
Transcript show: 'Hello'.  Transcript cr
```

will, when executed, display "Hello" in the transcript beginning at the current Transcript cursor position. The message cr will then instruct the Transcript to begin a new line. (Before executing this to try it, position the Transcript cursor at the beginning of a new line below the initial messages that are already there.) Note that it is easier to use cascading here:

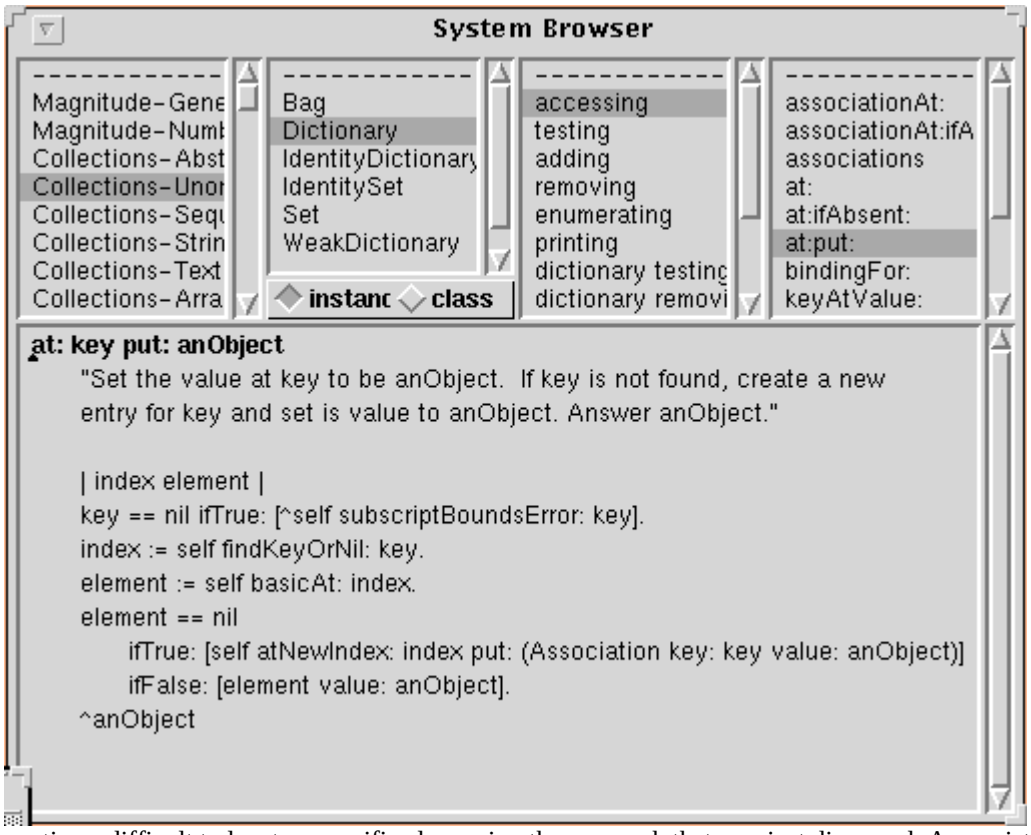
```
Transcript show: 'Hello'; cr
```

Displaying values of classes other than String can usually be done fairly easily by using the printString message to generate a string representation of a value. For example, try executing the code

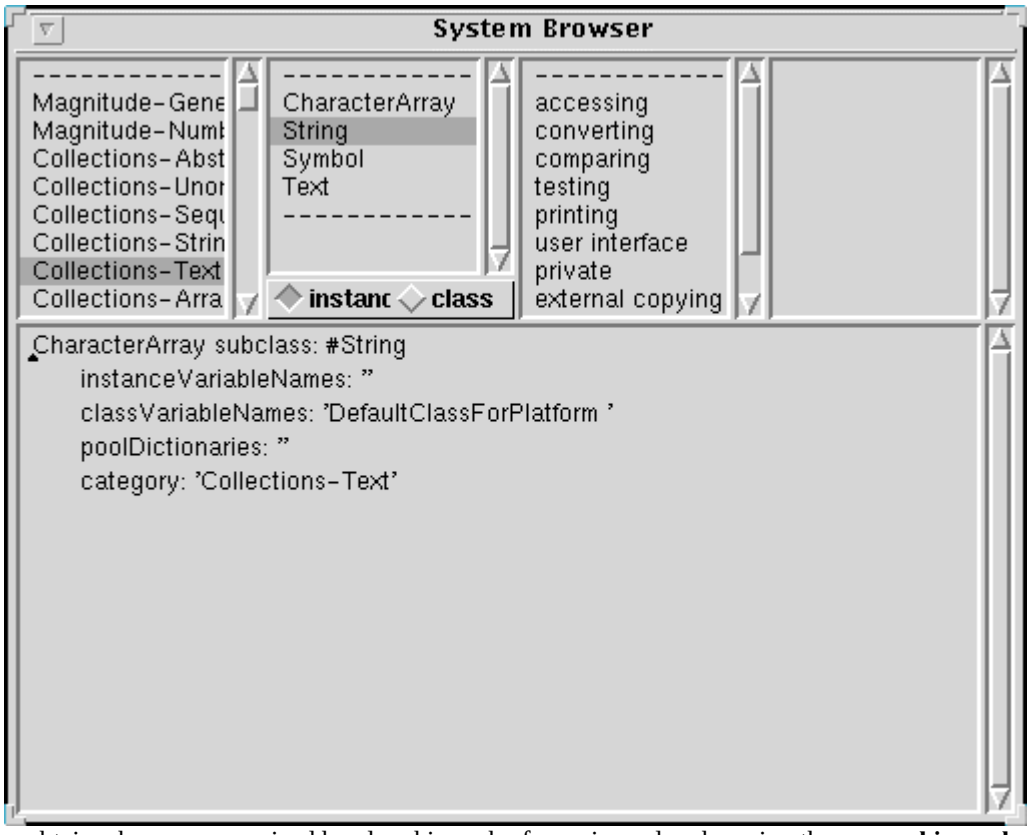
```
Transcript show: (5 + 9) printString; cr
```

Editing in VisualWorks Windows
Editing in a VisualWorks window is done by using procedures that are fairly standard for screen-based editors. Text that is typed is inserted at the cursor position. Replacement of text can be done by selecting the text (by dragging the mouse across it, or double-clicking to select a word, etc.), and then using the Backspace key to delete it or just typing its replacement to replace it. Cursor movement can be done using the arrow keys or by selecting the new cursor position with the mouse.
The scroll bars at the right side of a window can be used to scroll up and down, and a scroll bar at the bottom can be used to scroll left and right. Windows can be moved or resized in standard ways with the mouse at any time.

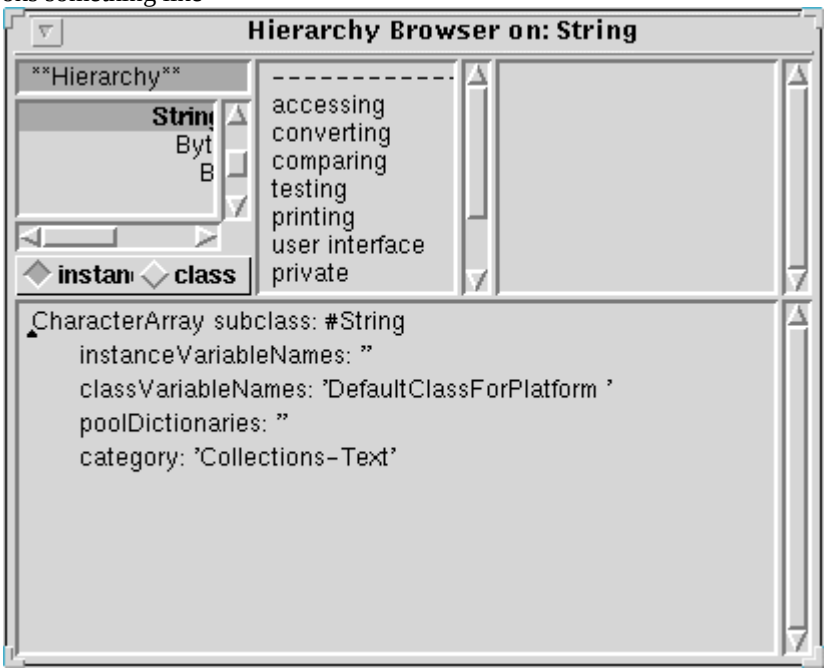
Using a Browser
A browser can be used to inspect the definition, comments, and code for all catagories, classes, and methods in the current image, both those that are provided in the initial image (i.e., the "built-in" classes and methods) and those that are added by the VisualWorks user. We will illustrate some of the uses of a browser in this section.
Open a browser from the Launcher with a **Browse-All Classes** selection or by using the shortcut button  . The classes are listed by category in the top left sub-window (the Category View). Select "Collections-Unordered" and the classes in this category will be listed in the next sub-window (the Class View). Select "Dictionary" and the protocols for the methods in class Dictionary will be shown in the next sub-window (the Protocol View). Select "accessing" from the Protocol View and the methods for this protocol will be listed in the rightmost sub-window at the top (the Method View). Finally, select "at:put:" in the Method View, and the code for the at:put: method is displayed in the bottom window (the Code View). Your browser window should now look like this:



It is sometimes difficult to locate a specific class using the approach that was just discussed. Any existing class can be found quickly by using the **find class...** selection from the [Operate] menu in the Category View (top left window of the browser). Select the **find class...** option and a dialog box will appear. Type the name of the class in this box (you can just type the name -- it will replace the highlighted text in the class name box), and then either press Return or select **OK**. Try this by typing String as the class name. Your browser should then look something like



We can obtain a browser organized by class hierarchy for a given class by using the **spawn hierarchy** menu selection in the Class View. Try this with class String selected, and you should get a new Hierarchy Browser that looks something like



The indented listing in the Class View of a Hierarchy Browser (there is no Category View in a Hierarchy Browser) indicates the superclass-subclass hierarchy for the class on which a hierarchy browser was spawned (String in this case). For example, we can see here that String is a subclass of CharacterArray, which is a subclass of ArrayedCollection, etc. Also, String has subclasses ByteEncodedString, GapString, and Symbol.

A Hierarchy Browser can help us to find a given method for a class more easily than is generally possible with a standard System Browser. For example, suppose that we wanted to find the method size for class String. (This method returns the size of a string.) We begin with a Hierarchy Browser on String and note that there is no size method in the accessing protocol (nor any other protocol). Selecting the superclass, CharacterArray, we see that there is also no size method in this class. Continuing up the inheritance hierarchy to ArrayedCollection, we find a size method here. So String instances inherit the size method from ArrayedCollection.
You can close the Hierarchy Browser using the [Window] **close** selection.

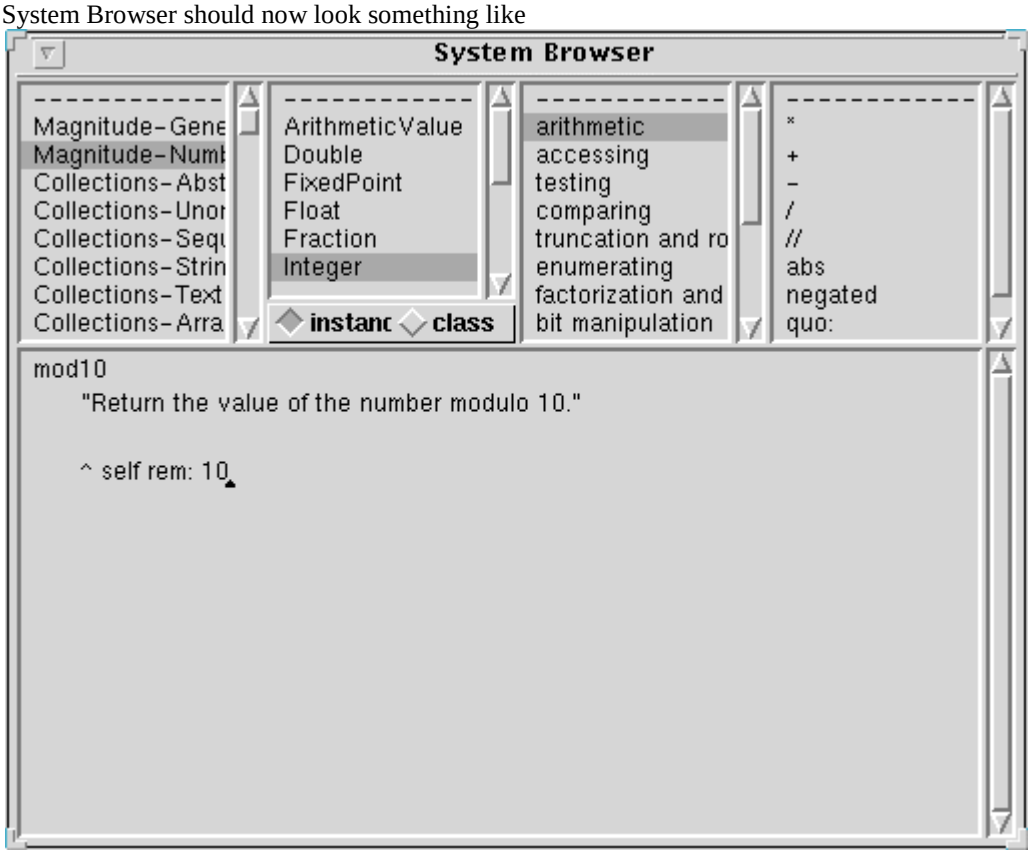
Adding a New Method
In this section we illustrate how a new method can be added to those in the current image. We will add a method "mod10" to the Integer class that will return the value of an Integer modulo 10. That is, for an Integer n,

```
n mod10
```

will have the value n rem: 10.
Select the Magnitude-Numbers category, the Integer class, and the arithmetic protocol in the System Browser. The arithmetic methods will be listed in the Method View, and a template for a method will be shown in the Code View. We will modify the template to produce the code for our new method.
First, select the first line of text ("message selector and argument names") in the Code View and replace it by the name of our new method (mod10). Then modify the documentation comment to indicate the function performed by the method. Finally, replace the temporary variable declaration and statements part by the code for our mod10 method:

```
^ self rem: 10
```

Your System Browser should now look something like



Select **accept** from the [Operate] menu in the Code View and the method will be compiled and added to the system. It will appear in the methods list of the Method View.
Test the mod10 method by executing (**do it**) some statements such as

```
Transcript show: (27 mod10) printString; cr
```
(This should cause 7 to be displayed in the Transcript.)

Adding New Classes or Methods From External Files

Classes, methods, or other code can be entered into the VisualWorks system by using the **file in** selection from various [Operate] menus. A file that is filed in must be in an external file format, which uses exclamation points to delimit class definitions, protocols, and methods. (This is the same format as is used for top-level input to GNU Smalltalk.)

We will illustrate the use of **file in** by implementing methods print and printNl (which are similar to methods of the same names in GNU Smalltalk) to make it easier for us to display results in the Transcript. The method "print" will cause its receiver to display its printString in the Transcript without a newline (cr) and "printNl" will cause its receiver to display its printString followed by a newline.

Create a file named "print.st" in the directory from which you started VisualWorks, and put the following text in the file:

```
!Object methodsFor: 'printing'!

print
        "Display the object in the transcript window;
         leave the cursor at the end of the object's print string."

        ( self isMemberOf: ByteString )
                ifTrue: [Transcript show: self]
                ifFalse: [Transcript show: self printString]!


printNl
        "Display the object in the transcript window, and start a new
line"

        self print.
        Transcript cr ! !
```

This code implements print and printNl as methods for class Object. Thus all classes will inherit them. (The test for a string in method print is done because the printString for a String inserts apostrophes around the String value. You can see this by executing code such as

```
Transcript show: 'Hello' ; cr; show: 'Hello' printString; cr
```

in a workspace, which will display

```
Hello
'Hello'
```

in the Transcript.)

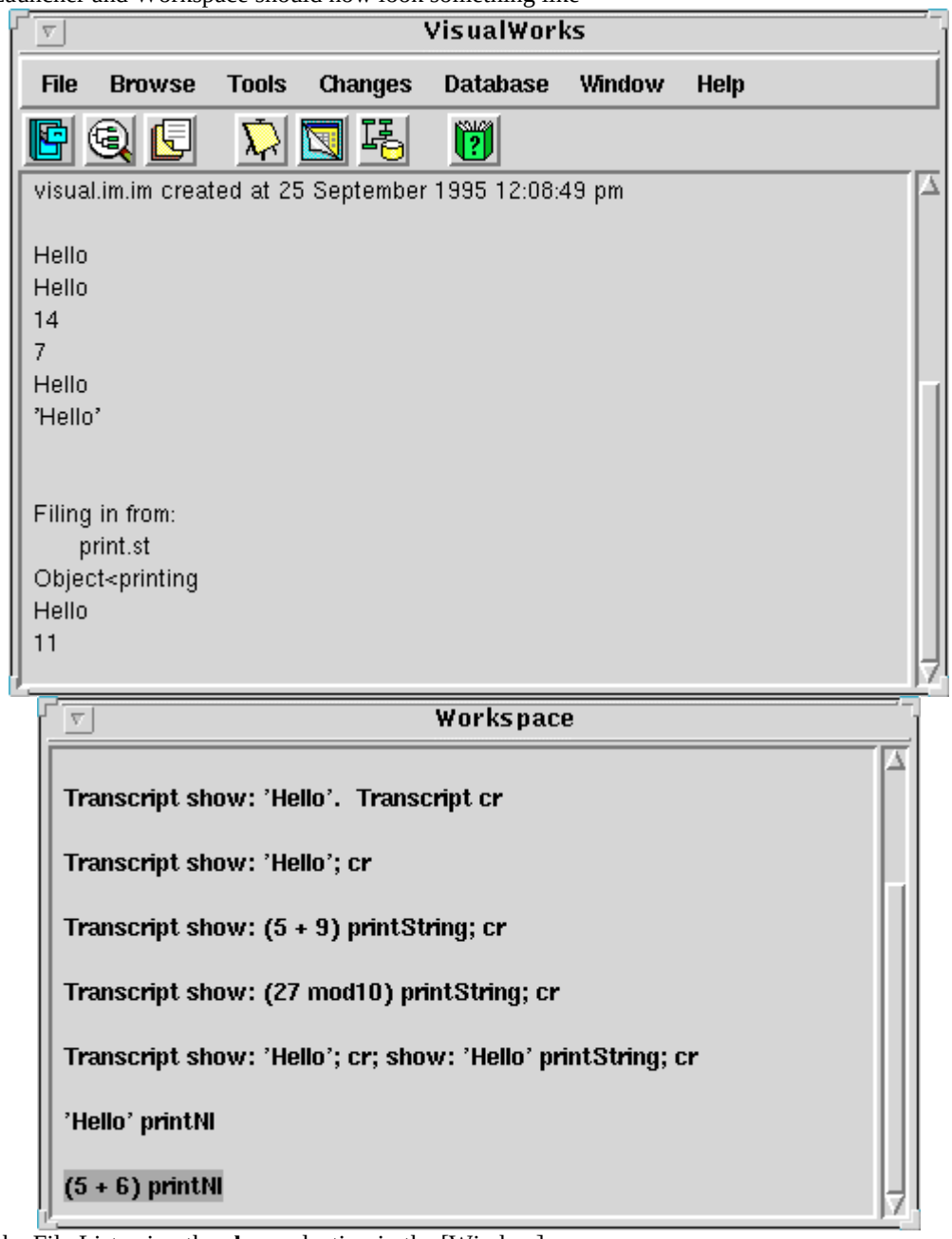
The easiest way to **file in** an external file is to use a File List, as was illustrated in Chapter 2. Open a File List from the **Tools-File List** option in the Launcher or by using the shortcut button  . In the first input field (the Pattern View) enter * and then Return, so that all the files in the local directory will be listed. Select the file print.st from the Names View, and the contents of the file that you created will appear in the bottom (File Edit) window. (Note: You can also use the File Edit window to create and edit files. Editing options are included in the File Edit [Operate] menu.)

Load the methods that are defined in the file print.st into VisualWorks by selecting **file in** from the Names View [Operate] menu. As the file is compiled, messages will be displayed in the Transcript indicating what is happening. If an error (syntactic or semantic) occurs, then the **file in** terminates. You can correct the error by editing the file in the File Edit window, saving it using the **save** option in the File Edit [Operate] menu, and filing it in again.

After successfully filing in print.st, you can test it by executing code such as

```
'Hello' printNl
```
and

```
(5 + 6) printNl
```
Your Launcher and Workspace should now look something like



Close the File List using the **close** selection in the [Window] menu.
Important Note: It is important to explicitly close each File List, rather than just exiting VisualWorks. On some systems, exiting VisualWorks without closing a File List will leave the File List running in a compute-bound mode, so that it will use every available cycle of cpu time even after the user has logged off.

Changing Existing Methods
Any method (or class) that is in the system can be changed (or removed) in much the same way as new code can be added. We will illustrate by changing the rem: method for Number to return a result that is 1 larger than the correct result.
Select the category Magnitude-Numbers, class Number, protocol arithmetic, and method rem: in the System Browser. The code for method rem: should be in the Code View. Change the line of code by appending "+ 1" to the end of the line:

```
^self - ((self quo: aNumber) * aNumber) + 1
```
Now before changing anything, set up a test in a workspace:

```
(27 rem: 5) printNl
```
and if you still have the mod10 method in your image a more interesting test is

```
(27 rem: 5) printNl. (27 mod10) printNl
```
Execute (**do it**) this code, and the correct answer(s) should be displayed in the Transcript:

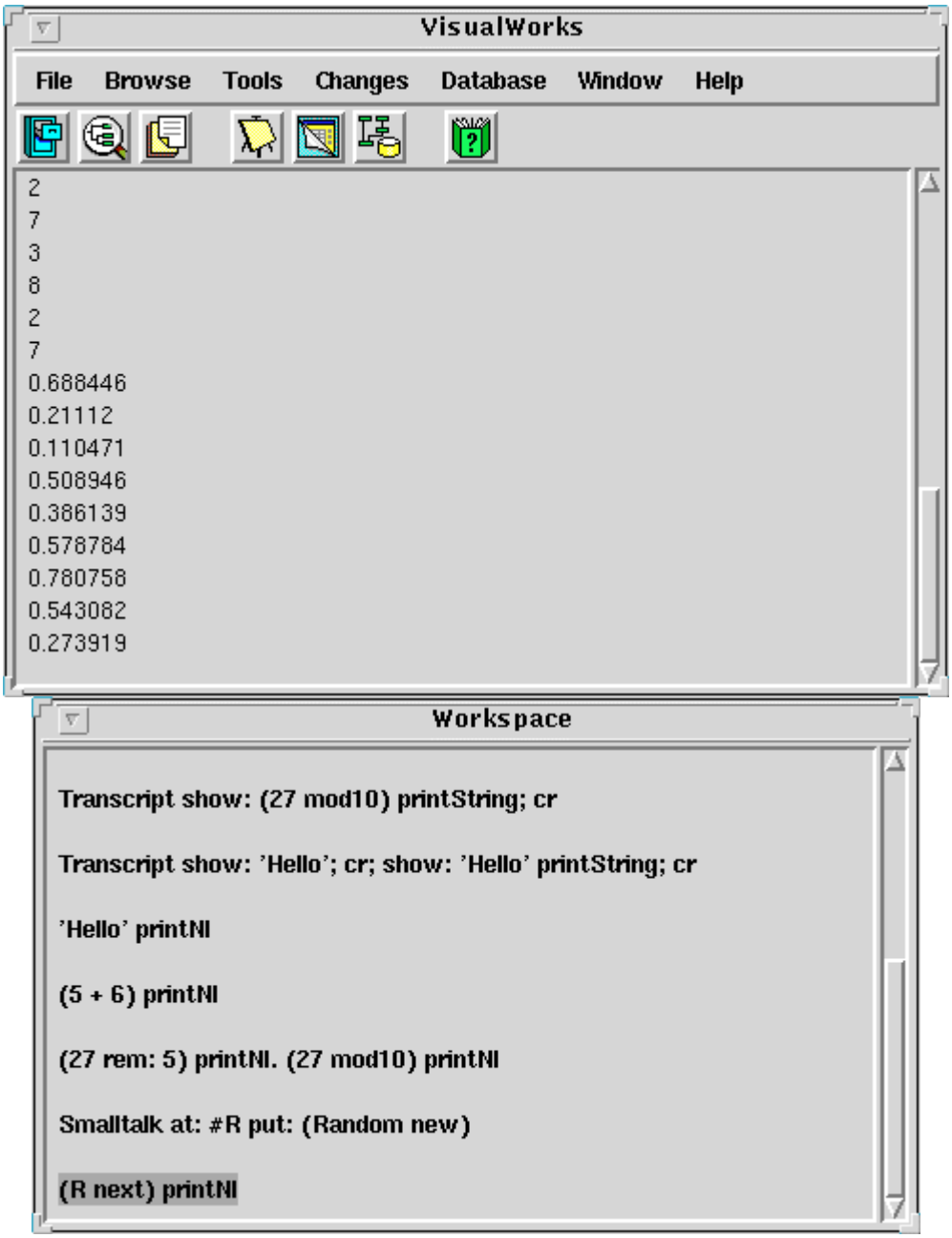
```
2
7
```
Now replace the rem: method by choosing **accept** from the [Operate] menu in the Code View of the System Browser. If there is no error indication, the new code for rem: has been compiled and entered into the system. To see this, execute the above code again, which will now give:

```
3
8
```
Remove the "+ 1" that was previously inserted into the code for rem:, **accept** the revised code, and test again to make sure that rem: now works properly.

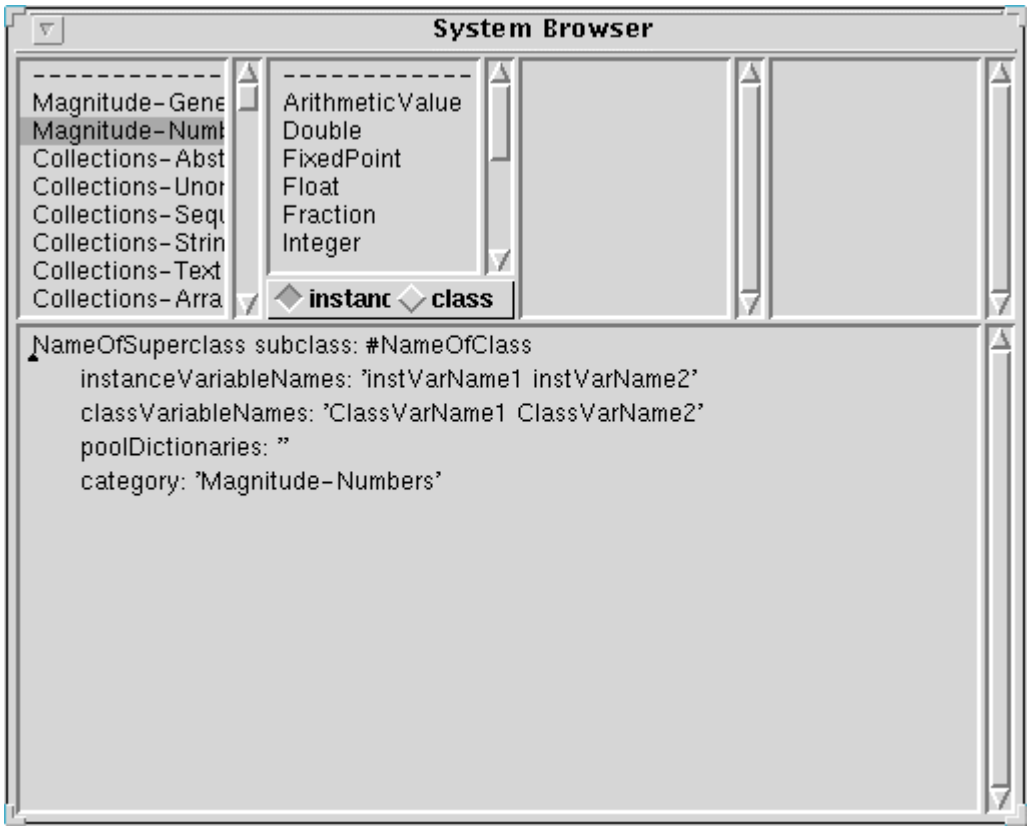
Adding a New Class
In this section we illustrate how to add a new class using a System Browser. (This is the intended way in which classes and methods are to be added.)
We will add a new class "Random2" as a subclass of existing class Random. An instance of class Random returns random numbers in response to the message "next". To see how this works, instantiate a random number by executing code such as

```
Smalltalk at: #R put: (Random new)
```
Now generate and display in the Transcript several random numbers by executing

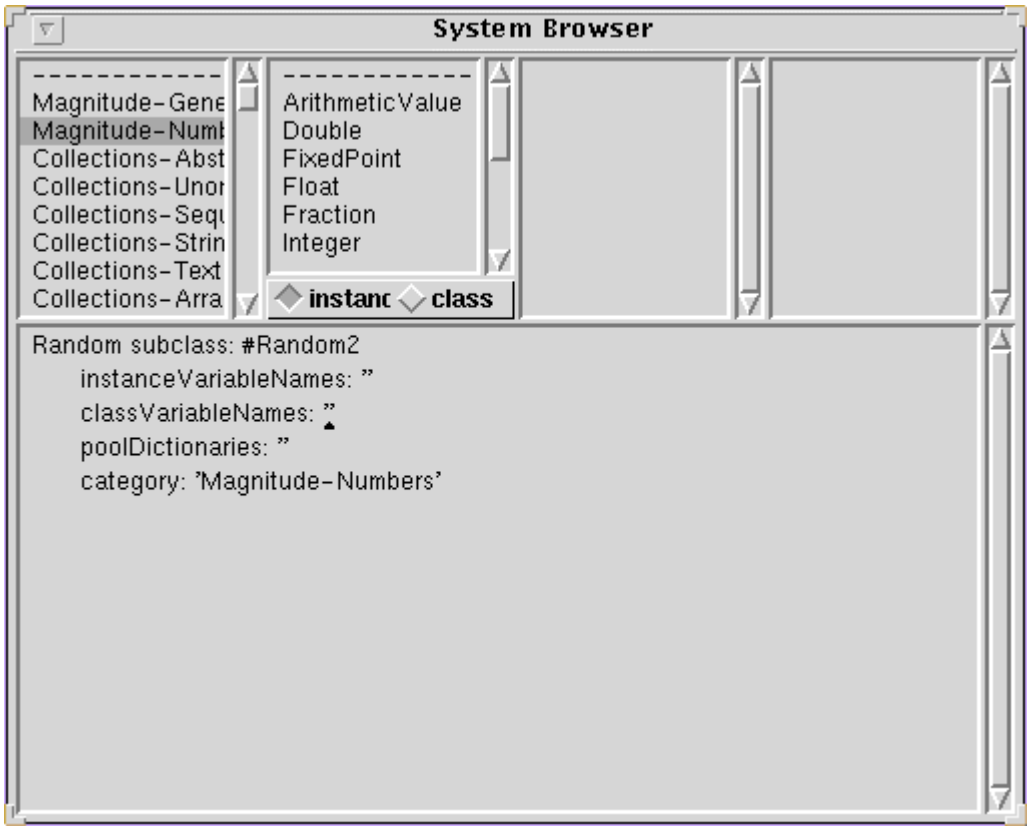
```
(R next) printNl
```
several times. The result of R next is a random number (Float) between 0.0 and 1.0, so your Launcher and Workspace should now look something like



(The random numbers in your Transcript will probably be different from those shown here.)
We will implement a new class, Random2 as a subclass of Random, where Random2 will also include a method between:and: to return a random integer between two given integer values. (Note that we could just as well have just added the between:and: method to class Random.)
In the system browser, select category Magnitude-Number with no class selection. There will then be a class template in the Code View:



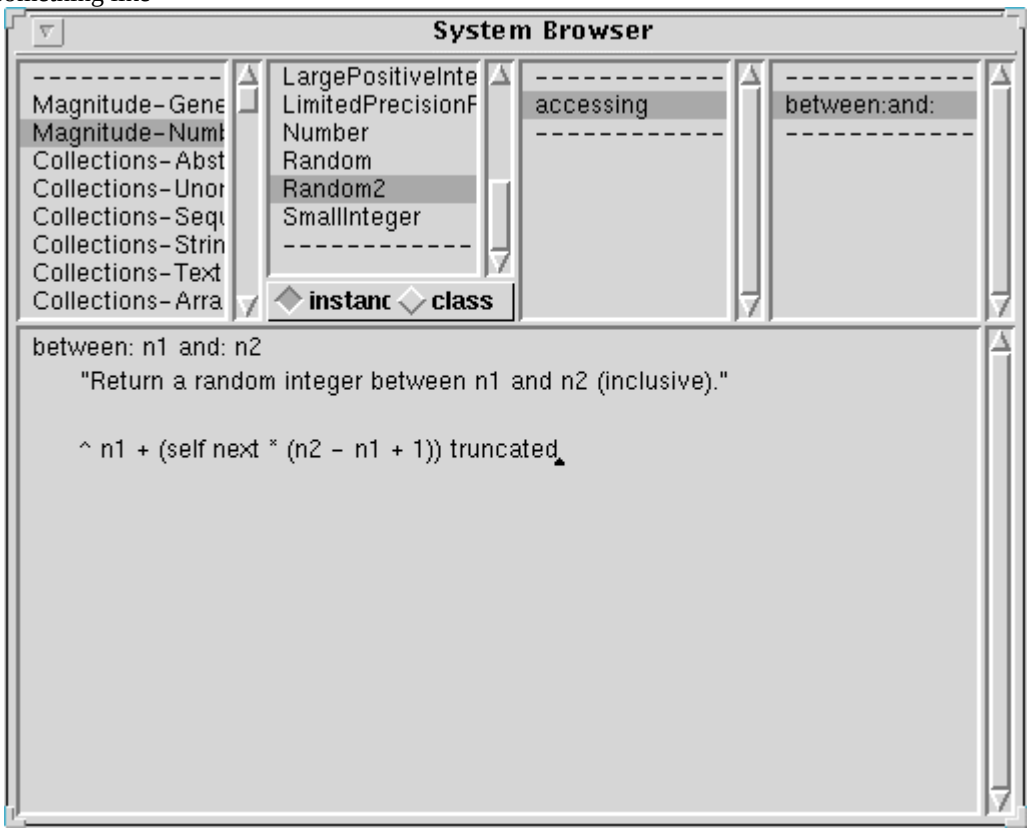
Edit the class definition template to define Random2 as a subclass of Random, with no instance variables nor class variables:



Compile the new class definition by using **accept** from the [Operate] menu of the Code View.
Next we add the method between:and: in protocol accessing of class Random2. First, add the protocol ("accessing") by choosing **add** from the [Operate] menu of the Protocol View. (Class Random2 should be selected in the Class View.) Type the new protocol name (accessing) into the dialog window and Return to record the new protocol.
Now edit the Code View window to contain the code for between:and:,

```
between: n1 and: n2
        "Return a random integer between n1 and n2 (inclusive)."

        ^ n1 + (self next * (n2 - n1 + 1)) truncated
```

and **accept**. The method name should appear in the Method View, and your System Browser should now look something like



We have now added the new class and method. Test it by executing code such as

```
Smalltalk at: #R2 put: (Random2 new)
```

and then execute the following several times:

```
(R2 between: 4 and: 11) printNl
```

This should display several random integers between 4 and 11 in the Transcript.

Saving Code into a File
As was briefly discussed in the previous lecture, the entire current image can be saved at any time, and later it can be used to restart VisualWorks from that saved state. However, an image is fairly large, and it is more efficient to save small modifications as external code files that can later be filed in to retrieve previous work.
To see how this works, we will save the Random2 class that was just added. From the System Browser with the Random2 class selected, choose **file out as...** from the [Operate] menu in the Class View. A dialog window should appear with the file name Random2.st highlighted. Change the file name if desired, then

select **OK** to file out the class. This file can later be filed in to reinstall the Random2 class, and this is left as an exercise for the reader.

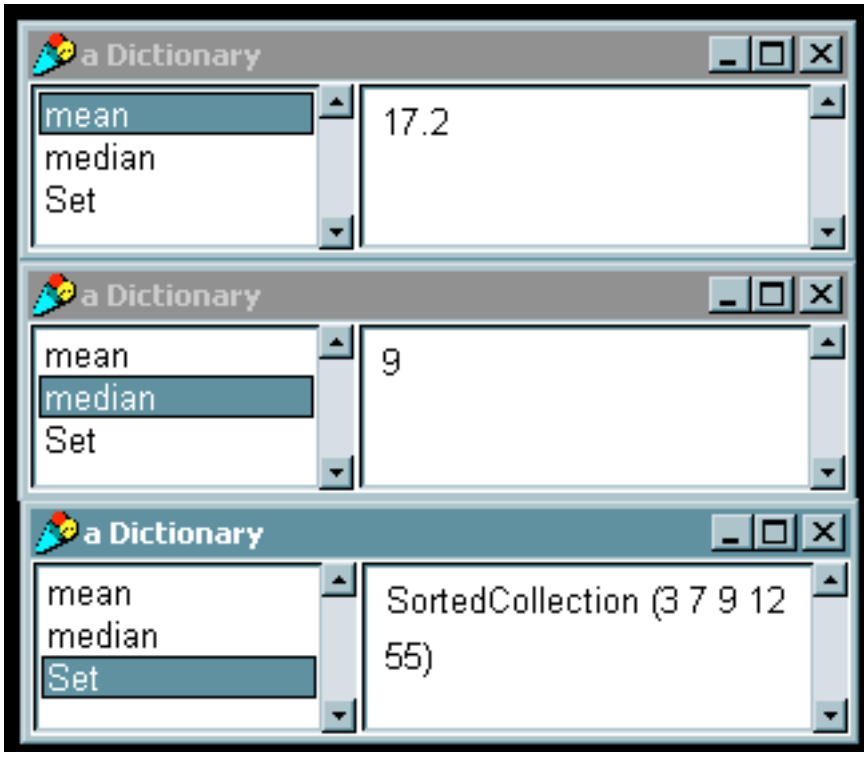
Lecture 15: System & Magnitude Classes

- **Overview**
  - Shared Object Protocols
    - Messages implemented for all objects
  - 3 Classes
    - Magnitude Classes
      - Numbers & characters
    - Collection Classes
      - Lists, Arrays, and Dictionaries
    - Streams
      - Text, Files, and Sockets
- **Shared Object Protocols**
  - 3 messages that can be applied to an object relating to its class
    - `class` finds out what class an object belongs to
      - `#(this is an array) class` ← Array
      - Similar to class are:
        - `isKindOf: aClass` returns true if aClass is a parent class of the reciever
          - `#(this is an array) isKindOf: Collection` ← true
        - `ismemberOf: aClass` returns true if the receiver is an instance of aClass.
          - `#(this is an array) isMemberOf: Collection` ← false
    - `isSequenceable` returns Boolean value depending on whether the receiver is created from a subclass of `SequenceableCollecetion`
      - #(this is an array) isSequenceable ← true.
      - `(Bag with: 'this' with: 'is' with: 'a' with: 'bag') isSequenceable` ← false
      - NOTE: class SequenceableCollection is called class IndexedCollection in smalltalk express, and isSequenceable is not available
    - `respondsToArithmetic: returns` Boolean
      - `respondsToArthithmetic` is implemented using the more general message, `respondsTo: aSymbol`, testing the symbols `#+, #-, #*,` and `#/`
  - `Comparing objects`
    - `==, ~~` CANNOT be overridden
    - `=, ~=` CAN be overridden
    - `isNil, notNil`
- Example: how to test and compare objects.
  - Suppose we want to write a method that takes a set, and creates a dictionary. The dictionary stores the sorted list of members, the median, and the mean.

```
compileStats: aSet
   |aDictionary sum setSize|
   aDictionary := Dictionary new.
   (aSet isKindOf: Set)
          ifFalse: [self notify: 'warning, argument is not a kind of
                 class Set'. ^nil].
   aSet class == SortedCollection
          ifTrue: [ aDictionary at: 'Set' put: aSet]
          ifFalse:
                 [ | aNewSet |
                 aNewSet := SortedCollection new.
                 aNewSet addAll: aSet.
                 aDictionary at: 'Set' put: aNewSet].
```

```
(aDictionary at: 'Set') do:
      [:x | x respondsToArithmetic
            ifFalse: [
                  self notify: 'Not numeric set'.
                  ^nil]].
setSize := (aDictionary at: 'Set') size.
aDictionary at: 'median' put: ((aDictionary at: 'Set') at:
      ((setSize/2) rounded)).
sum :=  0.
(aDictionary at: 'Set') do: [ :x | sum := sum + x].
aDictionary at: 'mean' put: ((sum/ setSize) asFloat).
^aDictionary.
```

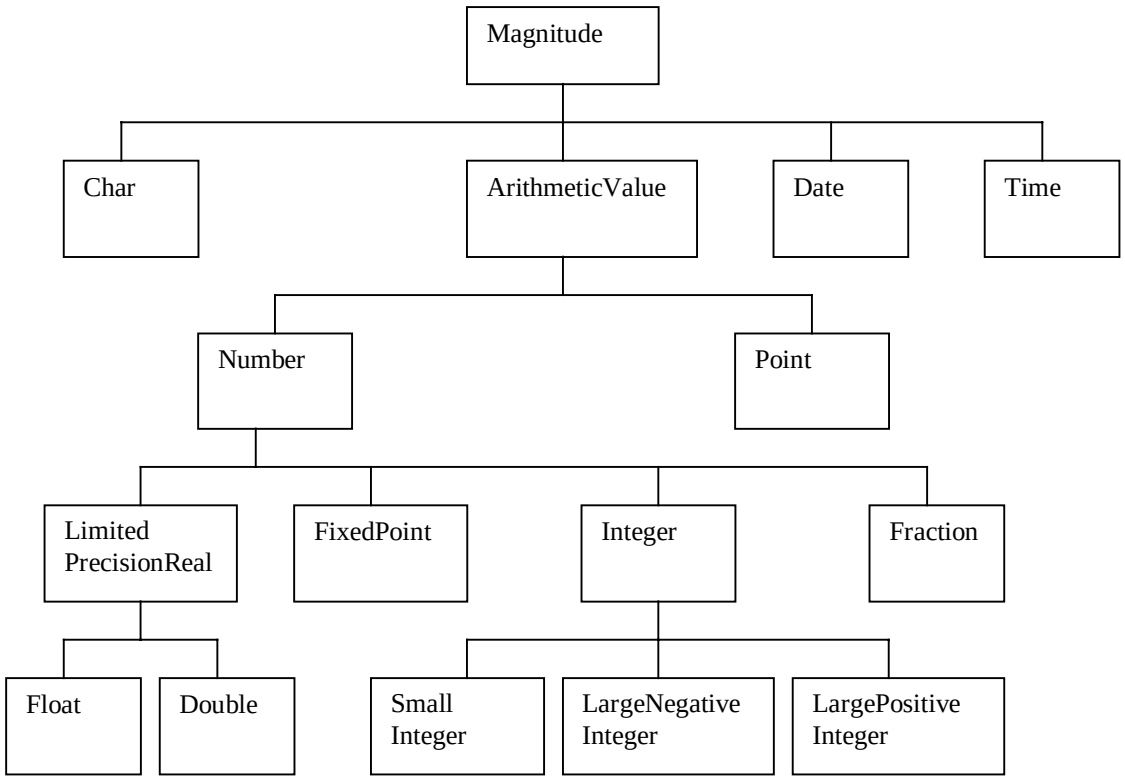
- Set(7, 12, 3, 9, 55) would result in the following dictionary



- **4 basic subclasses of the Magnitude class**
  - `Char`
    - Similar to `char` in C, basic class can be treated similarly to number
  - `ArithmeticValue`
    - Superclass for all numerical classes
  - `Date`
    - Very different from C style of date & time, comparable and human readable
  - `Time`
    - Very different from C style of date & time, comparable and human readable
- **Methods provided for comparison**
  - `aMagnitude between: oneMagnitude and: anotherMagnitude` (range comparison)
  - `aMagnitude max: anotherMagnitude` (max of the two magnitudes)
  - `aMagnitude min: anotherMagnitude` (min of two magnitudes)
  - `aMagnitude hash`
  - `<, <=, >, >=`

- **Example: More methods for complex numbers**

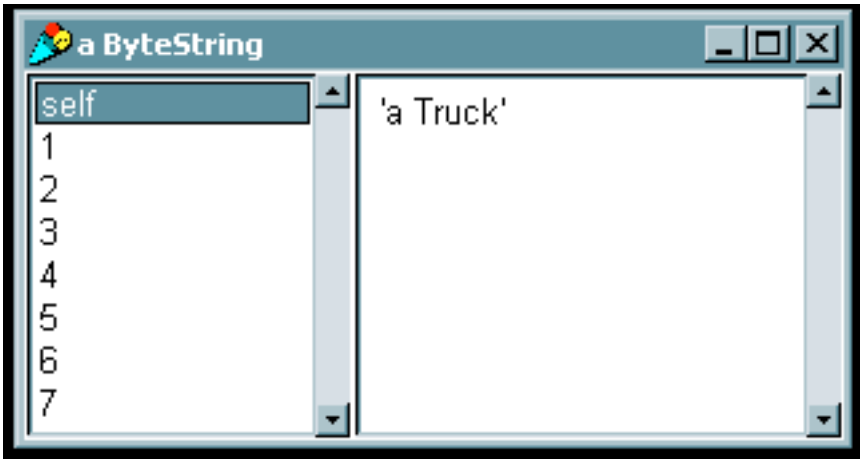
```
abs
        "Returns the absolute value of a complex number"
        ^(self realPart squared + self imaginaryPart squared)sqrt

< aComplex
  "Returns True if the reciever is less than aComplex"
  aComplex isKindOf: Complex
         ifTrue: [^self abs < aComplex abs]
         ifFalse: [^self error: 'Not a complex number'].

max: aComplex
       "Returns the greater value of aComplex and the receiver"
       self < aComplex
              ifTrue: [^aComplex]
              ifFalse: [^self].

= aComplex
  "Returns True if the receiver is equal to aComplex"
  aComplex isKindOf: Complex
         ifTrue: [
                ^self realPart=aComplex realPart and: [
                            self imaginaryPart = aComplex imaginary
                            part ]]
                      ifFalse: [^self error: 'Not a complex number']

  hash
         "hashes the absolute value of the reciver"
         ^self abs hash.
```



- **Partial Hierarchy**

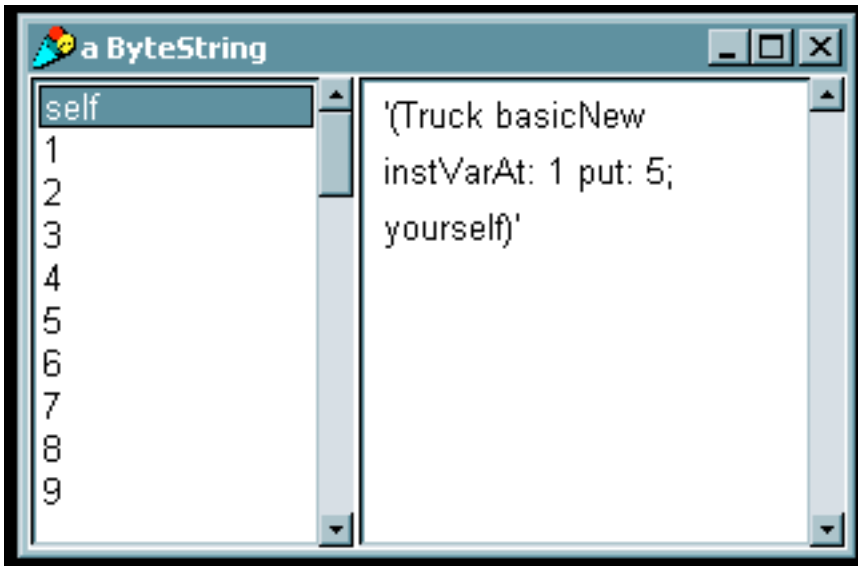
- **Type Conversion**
  - Converting to strings
    - To produce a string representation of an object use:
      - `objectName printString`

```
| aTruck |
aTruck := (Truck new) withSpeed: 5.
(aTruck printString) inspect.
```



      - `objectName storeString`

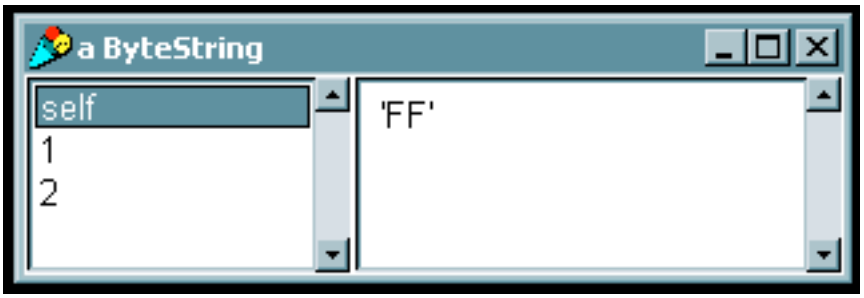
```
| aTruck |
aTruck := (Truck new) withSpeed: 5.
(aTruck storeString) inspect.
```



    - To produce the string representation of a number, the above can be used, or more specialized methods may be used
      - `anInteger printStringRadix: aRadix` (used for base aRadix representation)

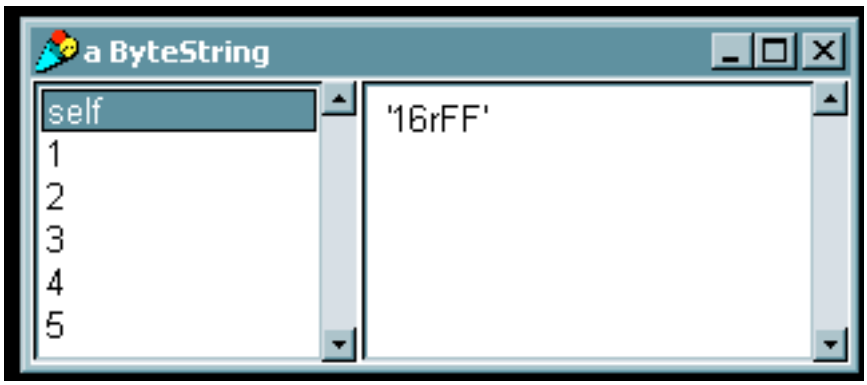
```
| anInteger |
anInteger := 255.
```

```
(anInteger printStringRadix: 16) inspect.
```



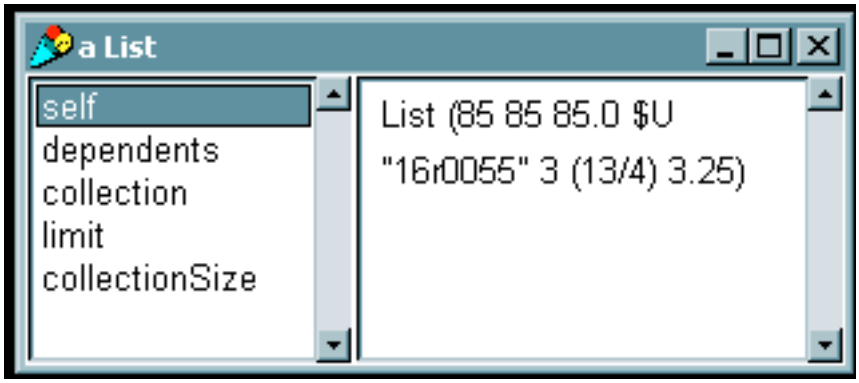
- `anInteger storeStringRadix: aRadix`

```
| anInteger |
anInteger := 255.
(anInteger storeStringRadix: 16) inspect.
```



- Converting strings to numbers
  - Requires streams to get strings from
    - This topic will be discussed in a later lecture.
  - Ex: Number `readFrom`: (ReadStream on: aStream)
- Type Conversion
  - Conversion is automatic and transparent
  - Conversion in direction integer -> fraction -> float to maintain accuracy
  - To explicitly do conversion use following methods
    - `asInteger`
    - `asFraction`
      - `asRational` in VisualWorks
    - `asFloat`
    - `asCharacter (integers only)`

```
| anInteger aFloat aList|
anInteger := 85.
aFloat := 3.25.
aList := List new.
aList add: anInteger asInteger.
aList add: anInteger asRational.
aList add: anInteger asFloat.
aList add: anInteger asCharacter.
aList add: aFloat asInteger.
aList add: aFloat asRational.
aList add: aFloat asFloat.
aList inspect.
```



- **Truncation, floor, ceiling and remainders**
  - Truncation done through `quo:` method
    - `11 quo: 5` => 2
    - `11 quo: -5` => -2
  - floor ceiling done though `//` operator
    - `11 // 5` => 2
    - `11 // -5` => -3
  - ceiling done through \\ operator
    - `11 \\ 5` => 3
    - `11 \\ -5` => -2
  - remainder is done through `rem:` method
    - `11 rem: 5` => 1
    - `11 rem: -5` => -1
- **Mathematical Operations**
  - Smalltalk provides basic subset of functions including
    - Trigonometry functions: `sin, cos, arcSin, arcCos`
    - Natural exponents and logarithms (`exp` and `ln`)
    - Exponents and logarithms
    - `gcd` and `lcd`
    - Ex:

```
55 gcd: 30 ← 5
6 lcm: 10 ← 30
0.523599 sin ← 0.5
6 exp ← 403.429
2.718284 ln ← 1
6 raisedTo: 3 ← 216
25 log: 5 ← 2
```

- **Date and Time**
  - Simple protocol for referencing and converting times & dates
  - Creating an time or date object
    - Use `now` method for creating the current time
      - `currentTime := Time now.`
    - Use `today` method for creating the current date
      - `currentDate := Date today.`
    - You can create an object with both current date and time
      - `rightNow := Date dateAndTimeNow.`
      - `rightNow := Time dateAndTimeNow`
    - Can create any time or date easily
      - `aDate := Date newDay: aDayOfTheYearInteger year: aYearInteger`
    - Time and Date Conversions
  - Timing execution and delays
    - Smalltalk provides a simple way to time the execution of a loop

```
| block1 block2 ms1 ms2 |
block1 := [100 timesRepeat: [Time now. Date today]].
ms1 := Time millisecondsToRun: block1.

block2 := [100 timesRepeat: [Time dateAndTimeNow]].
ms2 := Time millisecondsToRun: block2.
```

  - Smalltalk includes a similar class Delay. The Delay class is useful for creating timers. Timers can be used to update clocks or send messages regularly.
    - Delay should be used with the wait method
    - The following shows a simple clock, which writes to the Transcript.

```
[[true] whileTrue:
       [Transcript show: (Time now printString).
       (Delay forSeconds: 1) wait]] fork.
```

Lecture 16: The Collection Classes

- **Smalltalk's optimized Collection classes**
  - Unlike C, Smalltalk provides optimized classes for most types of collections
  - There are three types of Collections
    - Not keyed
      - Example: Bag
    - Keyed by integer
      - Example: Array, List, OrderedCollection
    - Keyed by value
      - Example: Set, Dictionary
  - For most situations, one of 5 types will suffice
    - SortedCollection
      - Sorts elements when inserted
      - Example returns SortedCollection ('a' 'b' 'c')

```
| aSortedCollection |
aSortedCollection := SortedCollection new.
aSortedCollection add: 'c'.
aSortedCollection add: 'a'.
aSortedCollection add: 'b'.
aSortedCollection inspect.
```

    - List
      - Most flexible, keeps elements in the order in which they were added.
      - Lists can be sorted.
      - Elements can be inserted anywhere
      - Example returns List ('a' 'b' 'c')

```
| aList aSortedList|
aList := List new.
aList add: 'c'.
aList add: 'b'.
aList add: 'a'.
aSortedList := aList sort.
```

    - Array
      - Does not require adding, removing, or sorting elements
      - Example returns #('d' 'b' 'c')

```
| anArray |
anArray := Array new.
anArray at: 1 put: 'a'.
anArray at: 2 put: 'b'.
anArray at: 3 put: 'c'.
anArray at: 1 put: 'd'.
```

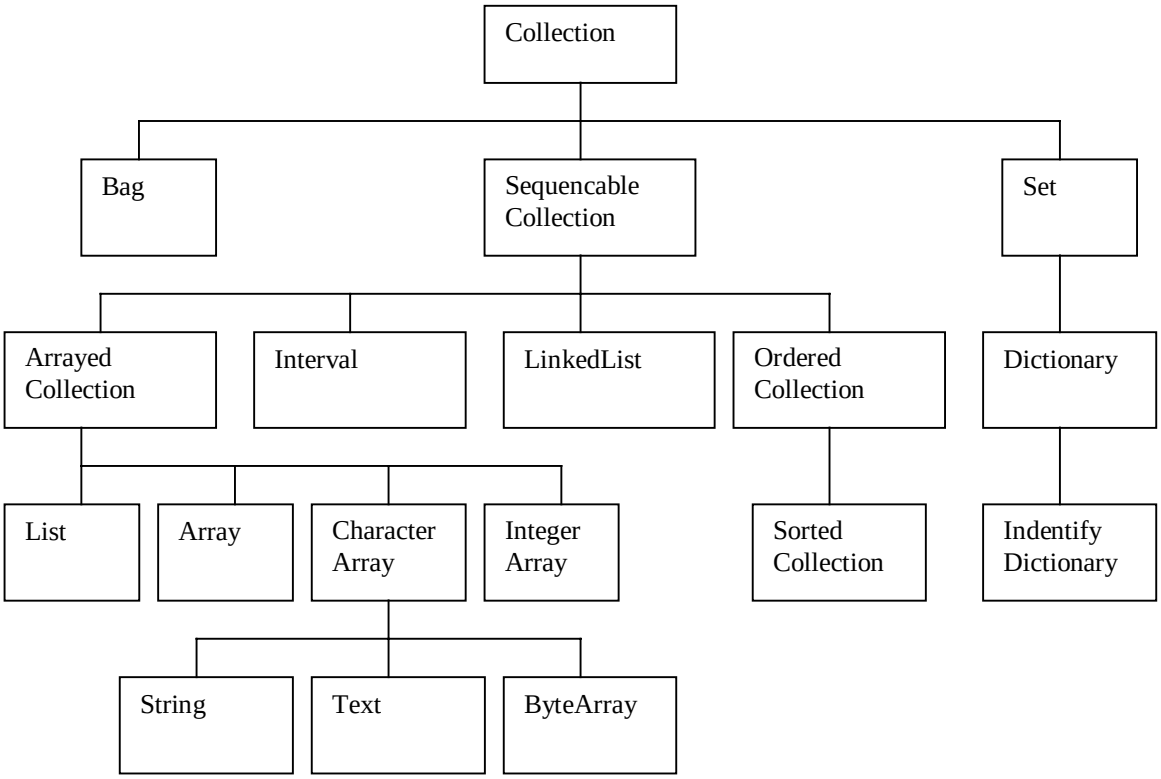
    - Set
      - Discards duplicate elements
      - Does not support replacing elements
      - Example
        - aSet ← Set ('a' 'b')
        - aList ← List ('a' 'b' 'a')

```
|  aList aSet |
aList := List new.
aList add: 'a'.
aList add: 'b'.
aList add: 'a'.
aSet := Set new.
aSet addAll: aList.
```

- Dictionary
  - New Concept to C programmers: Dictionary
- Otherwise known as Associated Hashtable
  - Add keys and values, and reference values through keys
  - Useful for global variables
  - Possible to associate keys with any kind of object
  - Ex:

```
| aThesaurus aCollection |
aCollection := Bag new.
aCollection add: 'big'.
aCollection add: 'enormous'.
aCollection add: 'huge'.
aThesaurus := Dictionary new.
aThesaurus at: 'large' put: aCollection.
aThesaurus at: 'small' put: 'little'.
```

- **Partial Hierarchy**



- **Iteration (what you can do with collections)**
  - Iterate over a collection
    - `do: []`
      - Ex: (sum ← 15)

```
| sum aCollection |
aCollection := Bag new.
aCollection add: 3.
aCollection add: 5.
aCollection add: 7.
sum := 0.
aCollection do: [ :x | sum := sum + x].
```

    - `reverseDo: []`

- Ex: (OrderedCollection(c b a))

```
| aReverseCollection aOrderedCollection |
aOrderedCollection := OrderedCollection new.
aOrderedCollection add: #a.
aOrderedCollection add: #b.
aOrderedCollection add: #c.
aReverseCollection := OrderedCollection new.
aOrderedCollection reverseDo:
       [:x | aReverseCollection add: x].
```

- `collect: []`
  - Useful to Create new collections from existing ones
  - Ex: (Bag(25 25 25… 0 0 0))

```
| someIntegers someNumbers|
someNumbers := Bag new.
1 to: 25 by: 0.2 do: [ :x | someNumbers add: x].
someIntegers := Bag new.
someIntegers := someNumbers collect:
        [ :x | x asInteger].
```

- Iterate over a collection and return a subset
  - `select: []`
    - Ex: (retuns only integers between 1 & 25 as floats)

```
| someIntegers someNumbers|
someNumbers := Bag new.
1 to: 25 by: 0.5 do: [ :x | someNumbers add: x].
someIntegers := Bag new.
someIntegers := someNumbers select:
        [ :x | (x // 1) asFloat  =  x].
```

  - `reject: []`
    - Ex: (returns only integers between 1 & 25 as floats)

```
| someIntegers someNumbers|
someNumbers := Bag new.
1 to: 25 by: 0.5 do: [ :x | someNumbers add: x].
someIntegers := Bag new.
someIntegers := someNumbers reject:
        [ :x | (x // 1) asFloat  ~=  x].
```

- Find occurrences of an object within the collection
  - `detect: []`
    - Ex: `#(a 'abc' 3 4 5) detect: [:x | x isInteger].` ← 3
    - Ex: `#(a 'abc' 3 4 5) findFirst: [:x | x isFloat] ifNone[nil]` ← nil
  - `findFirst: []`
    - Ex: `#(a 'abc' 3 4 5) findFirst: [:x | x isInteger].` ← 3
  - `findLast:`
    - Ex: `#(a 'abc' 3 4 5) findLast: [:x | x isInteger].` ← 5
- Perform special operations
  - `inject: into: []`
    - For using temp variables and initializing them outside the block
    - Ex: set the temp variable to 100

```
#(1 2 3) inject:100 into: [:x :y | x := x + y].
```
← 106

  - `with: do: []`
    - takes one object from the receiver and one from the argument
    - Ex: (result aBag= #('aA' 'cC'))

```
| aBag |
aBag := Bag new.
#('a' 'b' 'c') with: #('A' 'Z' 'C') do:
    [:x :y | x asUpperCase = y
        ifTrue: [aBag add: (x,y)]].
```

Lecture 17: An example using the Collection Classes

- Matrices are not provided by the collection classes, but can be added very easily. We will demonstrate the Collection classes by creating a Matrix class.
- The class should provide methods to add, multiply, and transpose matrices and scalars together.
- The matrix will be represented in row-major order, through a collection of rows, where each row is an ordered collection. To accomplish this, the matrix needs only two variables to keep count on the number of rows and columns
- The Class definition is straight forward

```
OrderedCollection variableSubclass: #Matrix2D
instanceVariableNames: 'numrows numcols '
classVariableNames: ''
poolDictionaries: ''
category: 'Examples-Matrix2D'

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Matrix2D
class instanceVariableNames: ''!
```

- The instance creation method creates a 2 dimensional matrix sized according to the arguments, and initializes all elements to 0.0. Note that the matrix is represented by aNumber1 OrderedCollections.

```
!Matrix2D class methodsFor: 'instance creation'!

rows: aNumber1 cols: aNumber2
      "Creates a 2D matrix of size aNumber1 X aNumber2 and
       initializes all elements to 0."

      | matrix |
      matrix := self new.
      1 to: aNumber1 do: [ :i | | temp |
            temp := OrderedCollection new.
            1 to: aNumber2 do: [ :j | temp addLast: 0.0 ].
                  matrix addLast: temp ].
      matrix setrows: aNumber1 cols: aNumber2.
      ^matrix! !
```

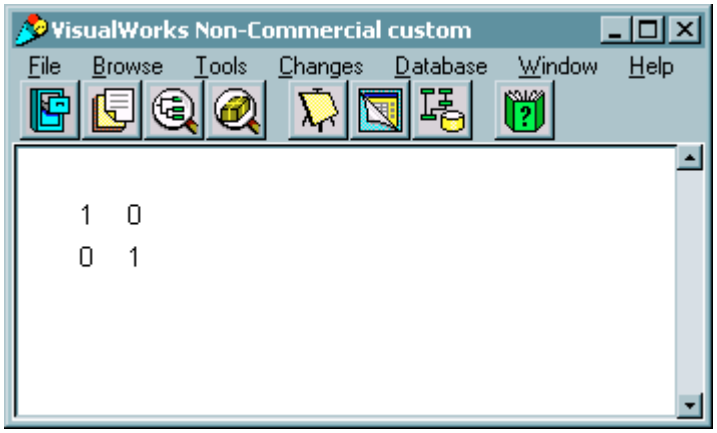
- The methods for accessing the matrix are straight forward as well, and are used only for getting elements and setting elements.

```
!Matrix2D methodsFor: 'accessing'!

at: anArray put: aNumber
   "Place an element (aNumber) in the row and column
    specified by anArray in the receiver."

   (self at: (anArray at: 1)) at:(anArray at:2) put:aNumber.!

atRow: aNumber1
   "Return an ordered collection from row aNumber1of the receiver."

   ^(self at: aNumber1).!

atRow: aNumber1 put: anOrderedCollection
   "Puts an OrderedCollection into the matrix as a row."
```

```
        super at: aNumber1 put: anOrderedCollection.!

    atRow: aNumber1 atCol: aNumber2
        "Return an element from row aNumber1, column aNumber2
         in the receiver."

        ^(self at: aNumber1) at: aNumber2.!

atRow: aNumber1 atCol: aNumber2 put: aNumber3
        "Place an element (aNumber3) in row aNumber1, column aNumber2
         in the receiver."

        (self at: aNumber1) at: aNumber2 put: aNumber3.!

cols
        "Returns the number of cols in the matrix."

        ^numcols.!

readAt: anArray
        "Returns an element from the row and column
         specified by anArray in the receiver."

        ^(self at: (anArray at: 1)) at:(anArray at:2).!

rows
        "Returns the number of rows in the matrix."

        ^numrows.!

setrows: aNumber1 cols: aNumber2
        "Sets the size of the matrix."

        numrows := aNumber1.
        numcols := aNumber2.! !
```

- To illustrate the access methods, we will create a 2x2 identity matrix with the code below. Recall an identity matrix is one which the top left to bottom right diagonal has 1 as the values of its elements, and all other values are 0.

```
| matrix1 |
matrix1 := Matrix2D  rows: 2 cols: 2.
matrix1 at: #(1 1) put: 1.
matrix1 at: #(1 2) put: 0.
matrix1 at: #(2 1) put: 0.
matrix1 at: #(2 2) put: 1.
matrix1 writeToTranscript.
```

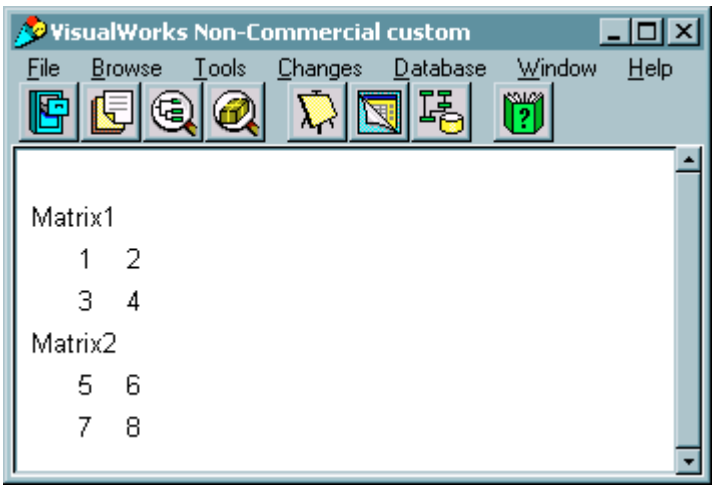


- The method `writeToTranscript`, as used above prints each row, an element at a time.

```
writeToTranscript
    "Writes the matrix to the Transcript."

    Transcript show: ' ';cr.
    1 to: (self rows) do: [ :i |
           Transcript show: ' '; tab.
           1 to: (self cols) do: [ :j |
                  Transcript show:
                         (self atRow: i atCol: j) printString; tab.
           ].
           Transcript show: ' ';cr.
    ].
```

- Although mathematical operations may appear to be complicated, the operations to be applied to the matrices are simple Collection manipulations.
- For the operation examples, the following matrices will be used. The code to create them is also shown below.

```
| matrix1 matrix2|
matrix1 := Matrix2D  rows: 2 cols: 2.
matrix1 at: #(1 1) put: 1.
matrix1 at: #(1 2) put: 2.
matrix1 at: #(2 1) put: 3.
matrix1 at: #(2 2) put: 4.
Transcript show: 'Matrix1'.
matrix1 writeToTranscript.
matrix2 := Matrix2D  rows: 2 cols: 2.
matrix2 at: #(1 1) put: 5.
matrix2 at: #(1 2) put: 6.
matrix2 at: #(2 1) put: 7.
matrix2 at: #(2 2) put: 8.
Transcript show: 'Matrix2'.
matrix2 writeToTranscript.
```

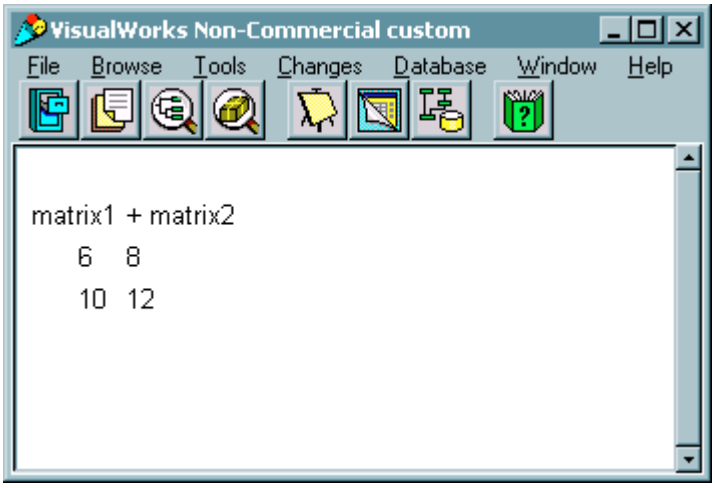


- matrixAdd: aMatrix adds aMatrix to the receiver and returns the sum of the two. A check is done to make sure both matrices are the same size

```
matrixAdd: aMatrix
   "Adds the receiver and aMatrix, that is, Receiver + aMatrix."

   | matrix |
   ( (numrows == ( aMatrix rows)) & (numcols == ( aMatrix cols)) )
   ifFalse:
            [ Transcript nextPutAll:
                  'matrixAdd - bad matrix size' ;endEntry.
         ^nil. ].
   matrix := Matrix2D rows: numrows cols: numcols.
   1 to: numrows do: [ :row |
         1 to: numcols do: [ :col |
               matrix atRow: row atCol: col put:
                     ((self atRow: row atCol: col) +
                         (aMatrix atRow: row atCol: col)).
         ].
   ].
   "Returns a new matrix"
   ^matrix
```

- Below is an example of adding two matrices.

```
Transcript show: 'matrix1 + matrix2'.
(matrix1 matrixAdd: matrix2) writeToTranscript.
```


VisualWorks Non-Commercial custom
File Browse Tools Changes Database Window Help
matrix1 + matrix2
6 8
10 12

- `matrixMult: aMatrix` multiplies the receiver and aMatrix and returns the product. A check is done to make sure the number of rows in the receiver is equal to the number of columns in aMatrix (rule of matrix multiplication).
- Recall the product of two matrices is as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \ldots & A_{1m} \\ A_{21} & A_{22} \ldots & A_{2m} \\ \vdots & & \\ A_{n1} & A_{n2} \ldots & A_{nm} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \ldots & B_{1m} \\ B_{21} & B_{22} \ldots & B_{2m} \\ \vdots & & \\ B_{n1} & B_{n2} \ldots & B_{nm} \end{bmatrix}$$

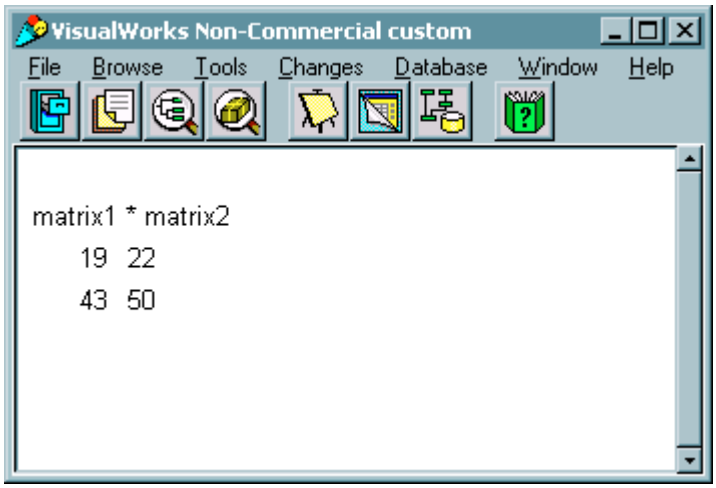
$$AxB = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} + \ldots + A_{1n}B_{n1} \ldots & A_{11}B_{1m} + A_{12}B_{2m} + \ldots + A_{1m}B_{nm} \\ A_{21}B_{11} + A_{22}B_{21} + \ldots + A_{2n}B_{n1} \ldots & A_{21}B_{1m} + A_{22}B_{2m} + \ldots + A_{2m}B_{nm} \\ \vdots & \\ A_{n1}B_{11} + A_{n2}B_{21} + \ldots + A_{nm}B_{n1} \ldots & A_{n1}B_{1m} + A_{n2}B_{2m} + \ldots + A_{nm}B_{nm} \end{bmatrix}$$

- The following code implements the equation above:

```
matrixMult: aMatrix
      "Multiplies the receiver and aMatrix, that is, Receiver *
aMatrix."

      | nrows ncols matrix sum |
      nrows := self rows.
      ncols := self cols.
      (ncols == ( aMatrix rows))
      ifFalse: [ Transcript nextPutAll:
          'matrixMult - bad matrix size' ;endEntry.
                  ^nil. ].
      matrix := Matrix2D rows: nrows cols:(aMatrix cols).

      1 to: (aMatrix cols) do: [ :k |
            1 to: nrows  do: [ :i |
                  sum := 0.
                  1 to: ncols do: [ :j |
                        sum :=  sum + ((self atRow: i atCol: j) *
                           (aMatrix atRow: j atCol: k)).
                  ].
                  matrix atRow: i atCol:k put: sum.
            ]
      ].
      ^matrix
```

- Below is an example of multiplying two matrices.

```
Transcript show: 'matrix1 * matrix2'.
(matrix1 matrixMult: matrix2) writeToTranscript.
```

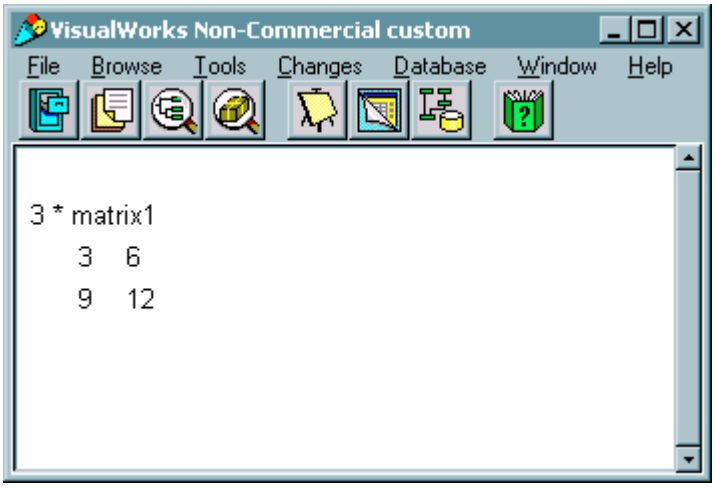


- The methods to add and multiply scalars are very similar to the matrixAdd: method, but even simpler.

```
scalarAdd: aNumber
   "Adds aNumber to each element of the receiver."

   | nrows ncols matrix |
   nrows := self rows.
   ncols := self cols.
   matrix := Matrix2D rows: nrows cols: ncols.
   1 to: nrows do: [ :row |
          1 to: ncols do: [ :col |
                 matrix atRow: row atCol: col put:
                        ( self atRow: row atCol: col ) + aNumber.
          ].
   ].
   ^matrix

scalarMult: aNumber
   "Multiplies each element of the receiver by aNumber."

   | nrows ncols matrix |
   nrows := self rows.
   ncols := self cols.
   matrix := Matrix2D rows: nrows cols: ncols.
   1 to: nrows do: [ :row |
          1 to: ncols do: [ :col |
                 matrix atRow: row atCol: col put:
                        ( self atRow: row atCol: col ) * aNumber.
          ].
   ].
   ^matrix
```

- Below is an example of multiplying a scalar and a matrix.

```
Transcript show: '3 * matrix1'.
(matrix1 scalarMult: 3) writeToTranscript.
```

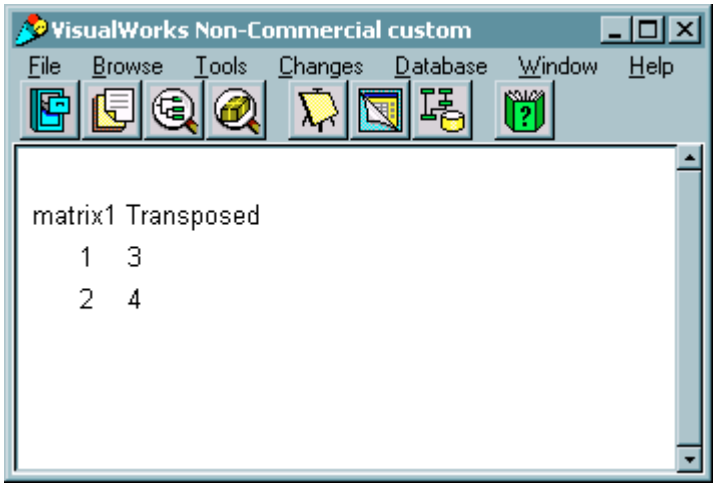


- The Transpose exchanges rows and columns.

```
transpose
	"Returns the transpose of the receiver."

	| nrows ncols matrix |
	nrows := self rows.
	ncols := self cols.
	matrix := Matrix2D rows: nrows cols: ncols.
	1 to: nrows do: [ :row |
		1 to: ncols do: [ :col |
			matrix atRow: row atCol: col put:
			    ( self atRow: col atCol: row ).
		].
	].
	^matrix
```

- Below is an example of transposing matrix.

```
Transcript show: 'matrix1 Transposed'.
(matrix1 transpose) writeToTranscript.
```



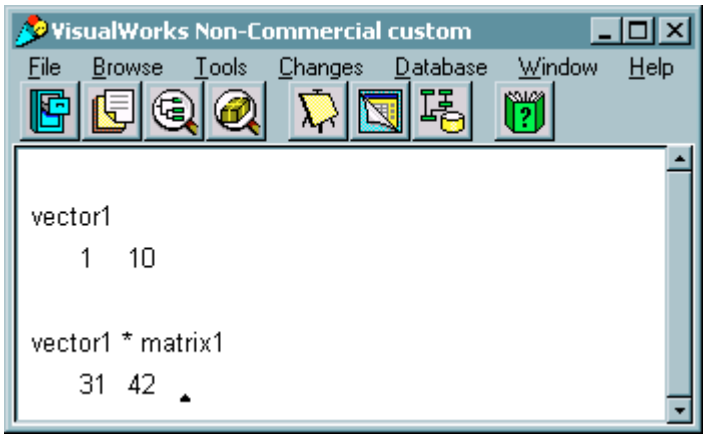
- Vectors are easily represented through the implementation of the Matrix class we have demonstrated, since a vector is nothing more than a single row of a matrix.
- Recall the product of a vector and a matrix is a vector as follows:

$$V = \begin{bmatrix} V_1 & V_2 \dots & V_n \end{bmatrix}, \quad M = \begin{bmatrix} M_{11} & M_{12} \dots & M_{1m} \\ M_{21} & M22 \dots & M_{2m} \\ \vdots & & \\ M_{n1} & M_{n2} \dots & M_{nm} \end{bmatrix}$$

$$V \times M = \begin{bmatrix} V_1 M_{11} + V_2 M_{21} + \dots + V_n M_{n1} \dots & V_1 M_{1m} + V_2 M_{2m} + \dots + V_n M_{nm} \end{bmatrix}$$

- The following code will create a vector and multiply it by matrix1

```
vector1 := Matrix2D rows:1 cols:2.
vector1 at: #(1 1) put: 1.
vector1 at: #(1 2) put: 10.
Transcript show: 'vector1'.
vector1 writeToTranscript.
Transcript show: 'vector1 * matrix1'.
(vector1 matrixMult: matrix1) writeToTranscript.
```



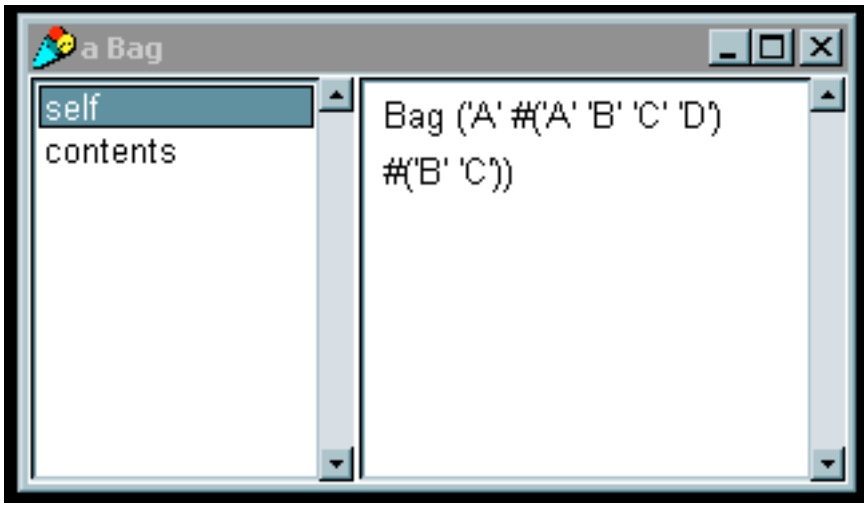
Lecture 18: The Stream Classes

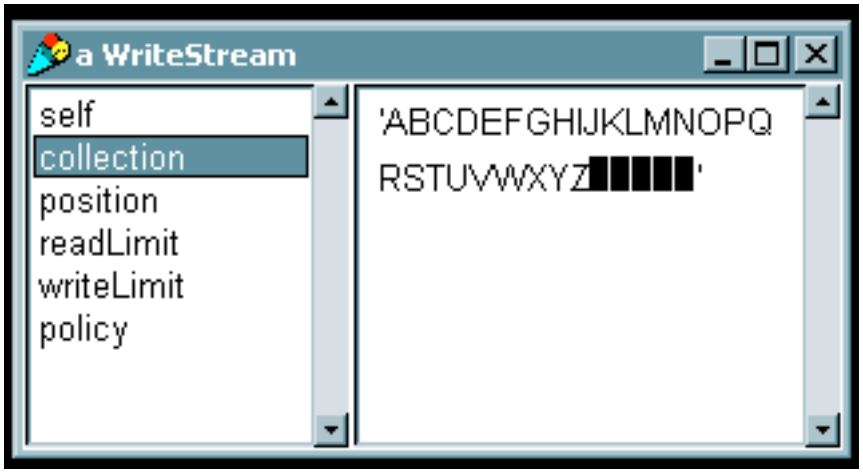
- **Streams**
  - Streams provide basic communication between the Virtual machine and the system
  - Types of streams
    - Semaphores
    - Sockets
    - Files
    - `stdin` & `stdout`
  - IMPORTANT: The programmer must close all open streams. Smalltalk will not close them for you, as in most compiled languages. The operating system has a limit on the number of open streams, and will quickly run out if the streams are not closed
- **Important methods for all Streams**
  - Accessing
    - `next` returns the next object in the stream
    - `next: anInteger` returns the next `anInteger` number of objects
    - `contents` returns all of the objects in the collection
    - `close` closes the stream
    - Ex:

```
| aStream anObject |
aStream := ReadStream on: #('A' 'B' 'C' 'D').
anObject := Bag new.
anObject add: aStream next.
anObject add: (aStream next: 2).
anObject add: aStream contents.
aStream close.
anObject inspect.
```

  - Writing
    - `nextPut: anObject` places `anObject` in the stream so that it is the next accessible
      - Ex: Generate & write the alphabet to a stream

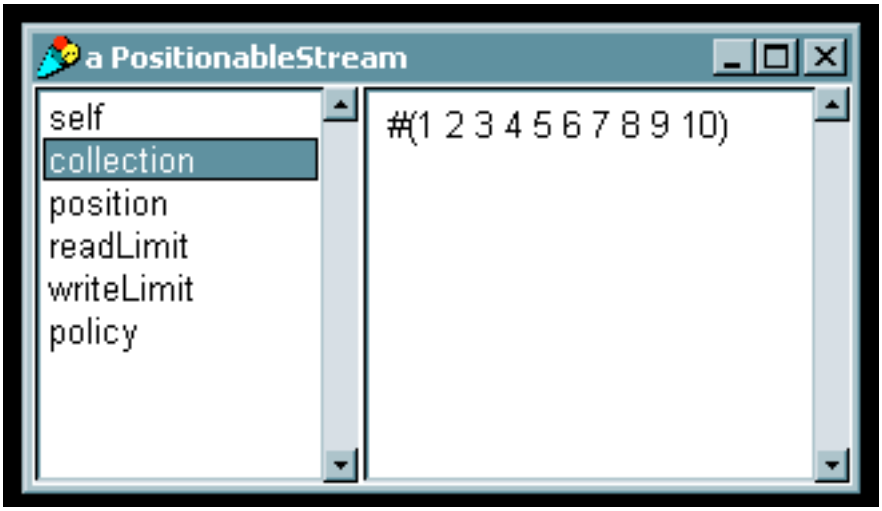
```
| aStream|
aStream := WriteStream on: (String new).
65 to: 90 do: [:aNumber | aStream nextPut: aNumber
asCharacter].
aStream close.
aStream inspect.
```





- `nextPutAll: aCollection` puts the contents of `aCollection` into the stream.
  - Example: Putting the number 1->10 into a Stream

```
| aStream aCollection |
aCollection := Array new:10.
1 to: 10 do: [ :aNumber | aCollection at: aNumber put:
aNumber].
aStream := ReadWriteStream on: (Array new: 10).
aStream nextPutAll: aCollection.
aStream close.
aStream inspect.
```



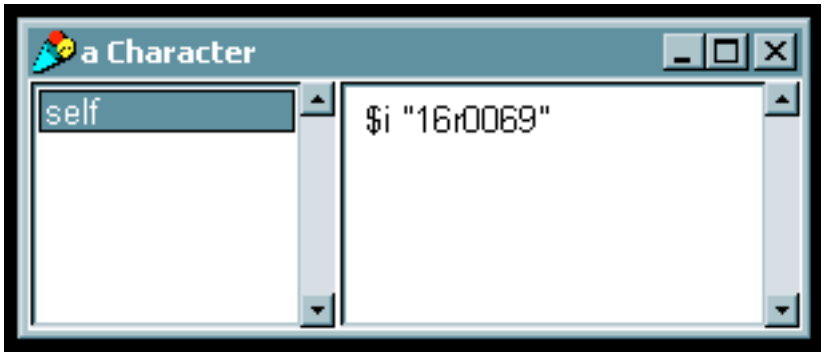
  - Example: Different way to get the same results. Which is the safer way? Which is the more “elegant” way?

```
| aStream aCollection |
aCollection := Array new: 10.
aStream := PositionableStream on: (aCollection).
1 to: 10 do: [ :aNumber | aCollection at: aNumber put: aNumber].
aStream close.
aStream inspect.
```

- **Important methods for Positionable Streams**
  - Accessing
    - `position` returns the position in the stream
    - `peek` returns the next object without advancing the position
    - `reset` resets the position in the stream

- `skip: anInteger` skips anInteger positions in the stream
- Ex:

```
| aStream anObject |
anObject := 'This is a single String'.
aStream := ReadWriteStream on: (String new).
aStream nextPutAll: anObject.
aStream reset.
aStream skip: 2.
(aStream peek) inspect.
```



- **Important methods for ReadStreams**
  - Instance Creation
    - `ReadStream on: aCollection`
  - All other positionable stream methods and general methods will work except for ones which write (such as at:put: methods)
- **Important methods for WriteStreams**
  - Instance Creation
    - `WriteStream on: aCollection`
  - Accessing
    - `flush` write all unwritten information to the stream
      - Good "book-keeping" habit to do before closing streams or saving images.
  - Similar to ReadStream, can access all methods of more general streams, but cannot read from streams
- **Important methods for External and File Streams**
  - Instance creation
    - 2 step process- make the filename, then apply the method to the filename. For the entire list of possible methods, refer to LaLonde, or the system browser under Filename->stream creation.

```
| aStream aFilename |
aFilename := Filename named: 'yourfile.txt'.
aStream := aFilename writeStream.
```

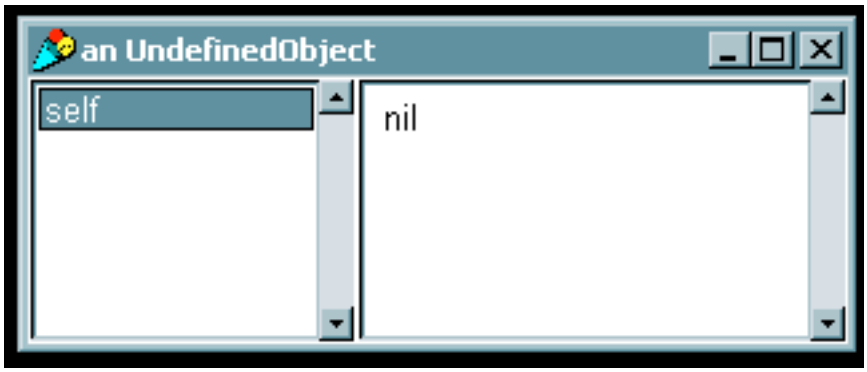
  - Accessing
    - `nextNumber: n` returns the next n bytes in the stream
    - `nextString` returns the next String from the stream.
    - `skipwords: nWords` advances the position nWords number of words (2 bytes, not to be confused with strings)
    - `wordPosition` returns the position in words
    - `wordPosition: wp` advances the position to wp in words
  - Writing
    - `nextPut: anObject` places `anObject` in the stream so that it is the next accessible

- `nextPutAll: aCollection` puts the contents of `aCollection` into the stream.
- Example, writing to a file.

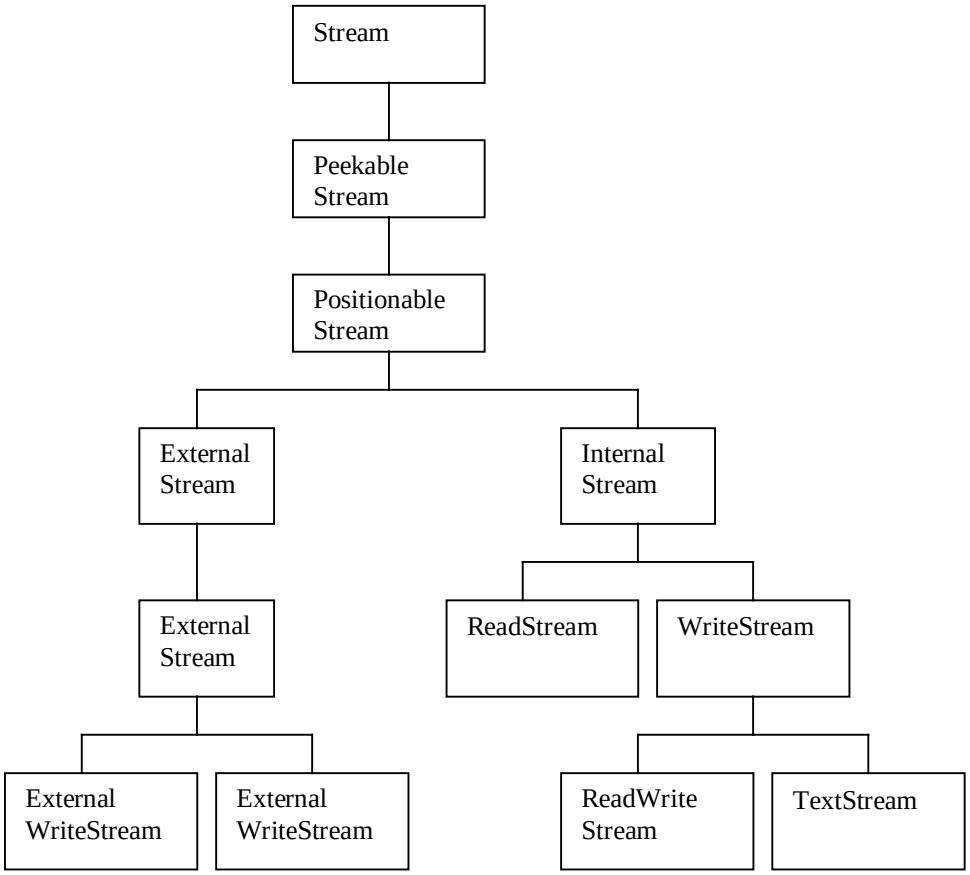
```
| aStream aFilename |
aFilename := Filename named: 'temp.txt'.
aFilename delete.
aStream := aFilename readWriteStream.
1 to: 20 by: 5 do: [ :aPosition |
    aStream wordPosition: aPosition.
    aStream nextPut: $D].
aStream close.
```

- **Common Mistakes**
  - Writing a collection to a stream is not the same as writing the contents of the collection.
    - Example: What is wrong with this? Shouldn't we see the number 2 instead of 'nil'? No, the first object is the collection, the second object is the end of the stream.

```
| aStream aCollection |
aCollection := Array new: 10.
aStream :=ReadWriteStream on: (aCollection).
1 to: 10 do: [ :aNumber | aCollection at: aNumber put: aNumber].
aStream reset.
(aStream peek) inspect.
```



- **Hierarchy**


Stream
Peekable Stream
Positionable Stream
External Stream
Internal Stream
External Stream
ReadStream
WriteStream
External WriteStream
External WriteStream
ReadWrite Stream
TextStream

Lecture 19: Matrix Example using Streams

- Recall the Matrix example. Now, rather than getting the matrix from standard in, we will read the matrix from a file. To maintain simplicity, we will keep the rules strict, but to allow for flexibility we will intelligently get the dimensions of the matrix. The rules of the file are as follows:
  - One matrix to a file
  - The matrix must be complete. That is, all rows must contain the same number of elements
  - The matrix will start with '[' and end with ']'.
  - Each row will be separated by a carriage return
  - Each element will be separated by white space (tabs, cr's, spaces).
  - No element may be negative
- So, using these rules, a 3x3 identity matrix would be represented as below:

```
[ 1 0 0
0 1 0
0 0 1 ]
```

- We need two methods, one to read from a file, and one to write to a file. We will add these methods to the Matrix2D class.
- The read method, `fromFile: aMatrix` is and instance creation method (like `new`). The method follows a simple parsing algorithm:

```
Get characters until '[';
rowCount = 1;
Get nextString;
    If nextString = '\n'
                increase rowCount;
                add Collection to Matrix
                reset Collection to nil
    If nextString is a number then add it to Collection
```

- This method accomplishes this by reading one character at a time, building up a string to be converted into numbers.
- Since we don't know how big the matrix will be, we can't store the elements immediately into the matrix. Instead, each row is read into an `OrderedCollection`.
- Once the `OrderedCollection` object is built, the `addLast`: method is called to add the `OrderedCollection` object to the matrix. The number of rows is then incremented.
- The following code implements the method as discussed above

```
fromFile: aName
   "Creates a 2D matrix from a file of the name aName"

   | aStream aFilename aString aChar aCollection aMatrix|
   aFilename := Filename named: aName.
   aStream := aFilename readStream.
   aChar := aStream next.

   "Create the Matrix"
   aMatrix := Matrix2D rows:0 cols:0.

   "eat up everything until the open bracket"
   [ aChar = $[ ]
         whileFalse: [ aChar := aStream next].
   "matrix has started"
```

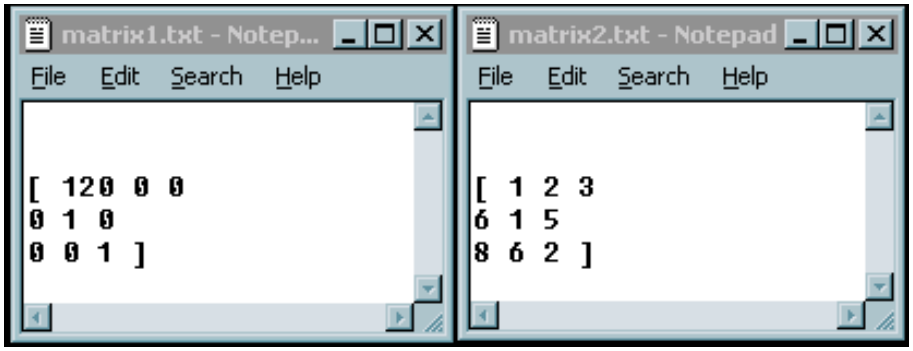
```
aCollection := OrderedCollection new.
aString := String new.
[ aChar = $] ]
  whileFalse: [
       aChar := aStream next.
       aChar asInteger = 13 "cr"
              ifTrue: [
                     aString size > 0
                       ifTrue: [
                         aCollection add:
                            (aString asNumber).
                         aString := String new.
                       ].
                     aMatrix addLast: aCollection.
                     aMatrix setrows: (aMatrix rows + 1)
                            cols: (aCollection size).
                     aCollection := OrderedCollection new.
              ].
              aChar isSeparator "any white space"
              ifTrue: [
              aString size > 0
                     ifTrue: [
                            aCollection add:
                              (aString asNumber).
                            aString := String new.
                     ]
              ].
       (aChar isDigit)
              ifTrue: [aString := aString,
                     (aChar digitValue printString)].
].
aStream close.

"Add the last one, since the ']' was on the last line"
aMatrix addLast: aCollection.
aMatrix setrows: (aMatrix rows + 1)
       cols: (aCollection size).
aCollection := OrderedCollection new.

^aMatrix
```

- The method to write the matrix to a file is considerably more simple. A '[' is written, then each row is written as characters, then a ']' is written.

```
writeToFile: aName
       "Writes the matrix to the file aName. The format is
   such that fromFile:    can be called to read it back
   into a matrix."

       | aStream aFilename|
       aFilename := Filename named: aName.
       aStream := aFilename writeStream.

       aStream nextPut: $[.
       1 to: (self rows) do: [ :row |
              1 to: (self cols) do: [ :col |
                     ((self atRow: row atCol: col) printString)
                 do: [ :char |
                            aStream nextPut: char
                     ].
                     aStream nextPut:$ . "space"
              ].
              row = (self rows)
```

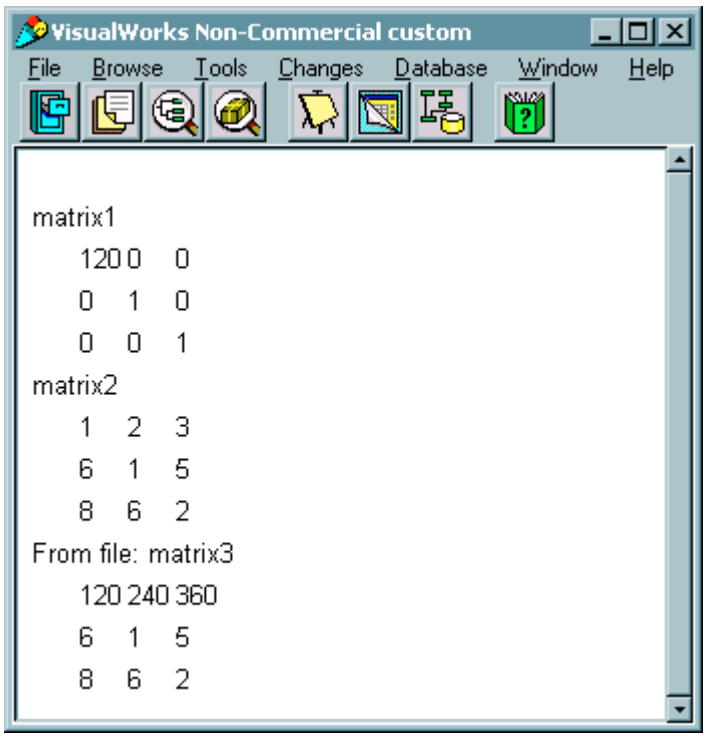
```
                ifFalse: [
                    aStream nextPut: 13 asCharacter. "cr"].
        ].
        aStream nextPut: $].
        aStream close.
```

- To illustrate the use of the new file methods, we will read two matrices from text files (“matrix1.txt” and “matrix2.txt”), then multiply them together. Their product will be written to a file (“matrix3.txt”). Below is a screen capture of the two input text files.

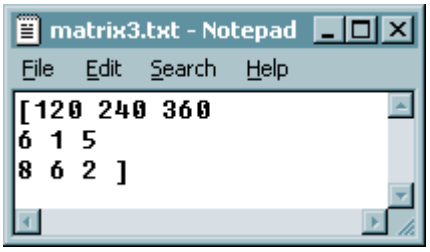


- The code below will now multiply the matrices together and write the product to a file. The code also reads the output file back in and prints it to the Transcript as a form of visual sanity check.

```
| matrix1 matrix2|
matrix1 := Matrix2D fromFile: 'matrix1.txt'.
Transcript show: 'matrix1'.
matrix1 writeToTranscript.
matrix2 := Matrix2D fromFile: 'matrix2.txt'.
Transcript show: 'matrix2'.
matrix2 writeToTranscript.
(matrix1 matrixMult: matrix2) writeToFile: 'matrix3.txt'.
Transcript show: 'From file: matrix3'.
(Matrix2D fromFile: 'matrix3.txt') writeToTranscript.
```



- The code results in the output file



Lecture 20: Dependency Mechanisms

- **Dependency**
  - Objects depend on the state of other objects.
  - For example, we have a lamp with a light switch and a light bulb
- When the state of the switch is changed, the light bulb is notified of the change
- This does not imply when the light bulb status changes (burns out) that the light switch is notified
  - There is no explicit relationship between two objects so we must find a way of connecting them. Through a special connection, one object is said to be "dependent" on the other object. Smalltalk has a dependency mechanism to handle the connections between two objects.
- Use collections to store groups of dependent objects.
  - Instead of a light switch, we now have a traffic light with three light bulbs. Each light has to notify all other lights when it turns itself on so they will turn themselves off. In this example, no two lights should be on at the same time.
  - The protocol for sending messages and updating is provided by class Object.
    - A Object sends itself a `changed` message & its dependents are informed automatically via the `update: aMessage,` where `aMessage` can be any message
      - `self changed`
        - dependents determine what was changed
      - `self changed: anAspect`
        - Object informs dependents of a change involving `anAspect`
      - `self changed: anAspect with: aValue`
      - `update: with: from:`
        - Method is inherited from class Object, but it is usually overridden.
  - Methods for adding and removing dependencies
    - `addDependent:`
      - adds the dependency of the argument's object to the receiver's object
      - `tire: addDependent: automobile.` The automobile is sent a message if the state of the tire is changed.
    - `removeDependent:`
      - removes the dependency of the receiver from the argument
      - the receiver no longer updates the argument
      - `tire: removeDependent: automobile.` The automobile is now not updated when the state of the tire changes
    - `release`
      - Releases all of the dependents of an object
    - `dependents`
      - Returns an Ordered Collection containing the dependents
  - Now we can look at the code for the lamp
- To allow for the light bulb to be easily changed, we'll give it a function `update: signal`
- The lamp object should also provide the methods for the following
  - Getting and setting its state
  - Showing the state by writing to the Transcript
  - Getting and setting its identification number (id)

```
Object subclass: #Lamp
       instanceVariableNames: 'state id'
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Examples-Lamps'!
```

```
state
	"Returns the state of the lamp: 0=off, 1=on."

	^state.!

state: anInteger
	"Sets the state of the lamp: 0=off, 1=on.
	The self changed message tells the lamps
	dependents it is now on."

	anInteger isZero ifFalse: [ state := 1.
					self changed: #on]
					ifTrue: [ state := 0 ].! !
id: anInteger
	"Sets the id of the lamp"
	id := anInteger.!

id
	"Gets the id of the lamp"
	^id.

showState
	“Shows the state of the lamp in the transcript”
	Transcript show: 'Lamp '; show: (id printString).
	Transcript show: ' state: '; show: (state printString); cr.

!Lamp methodsFor: 'updating'!

update: signal
	"If some other lamp has turned on, turn myself off."

	(signal = #on)
	ifTrue: [state := 0.
			self changed].! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Lamp class
	instanceVariableNames: ''!


!Lamp class methodsFor: 'instance creation'!

new
	"Gets a new instance."

	^super new! !
```

- Now we will look at the code for the lamplist and traffic signal. Keep in mind that the LampList and the Lamps are the objects doing the work, and the TrafficSignal provides a useful example of a device which could use the LampList.

```
	OrderedCollection variableSubclass: #LampList
		instanceVariableNames: ''
		classVariableNames: ''
		poolDictionaries: ''
		category: 'Lamps'!


	"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

	LampList class
		instanceVariableNames: ''!
```

```
!LampList class methodsFor: 'instance creation'!

new
	"Creates a new instance."

	^super new!

new: size
	"Creates a new instance."

	^(super new: size)! !

"-------------------------------------------"!

Object subclass: #TrafficLight
	instanceVariableNames: 'lamplist '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Examples-Lamps'!
```

- In the method `initialize`, each lamp is created, and added to the lamplist. Each lamp in the lamp list is then adds the other lamps to its list of dependents.

```
!TrafficLight methodsFor: 'initialization'!

initialize
	"Creates the three lights and turns the first on"

	| lamp index|
	lamplist := LampList new:3.
	Index := 1.
	3 timesRepeat:
			[ lamp := Lamp new.
			  lamp state: 0.
			  lamp id: index.
			  lamplist add: lamp.
	(lamplist at: 1) state: 1.
	1 to: lamplist size do: [ :l |
		1 to: lamplist size do: [ :dep |
			l = dep ifFalse: [
				(lamplist at: l) addDependent:
				(lamplist at: dep)]]].! !

!TrafficLight methodsFor: 'accessing'!

lamplist
	"returns the lamplist"
	^lamplist.!
```

- The method gets the lamp that is on, then turns on the next lamp. If the third lamp was on, then the first lamp is turned on. Each time the light is changed, a message is written to the transcript to log that the light was changed.

```
changeLight
	"advances to the next light in the list"
	| index |
	Transcript show: 'Changing the Lights'; cr.
	index := (self lightIsOn) id.
```

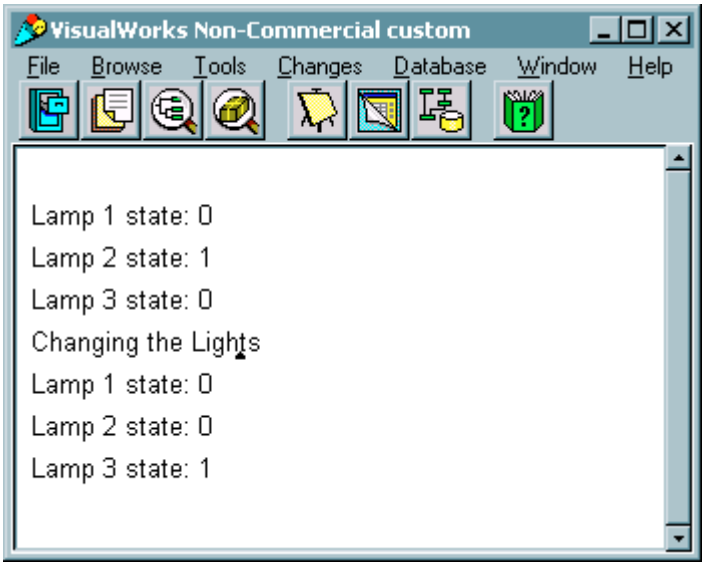
```
        (lamplist at: ((index rem: 3) + 1) state: 1.
```

- The method `lightIsOn` is used to tell the traffic light which lamp is currently turned on. It finds the first lamp which is on (only 1 should be on), and returns that light.

```
lightIsOn
        "returns the index of the light that is on."

        ^(lamplist detect:
                [ :lamp | lamp state = 1]).

showStates
        lamplist do: [ :lamp| lamp showState].

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

TrafficLight class
        instanceVariableNames: ''!


!TrafficLight class methodsFor: 'instance creation'!

new
        "Creates a new instance and initializes the lights."

        ^(super new) initialize.! !
```

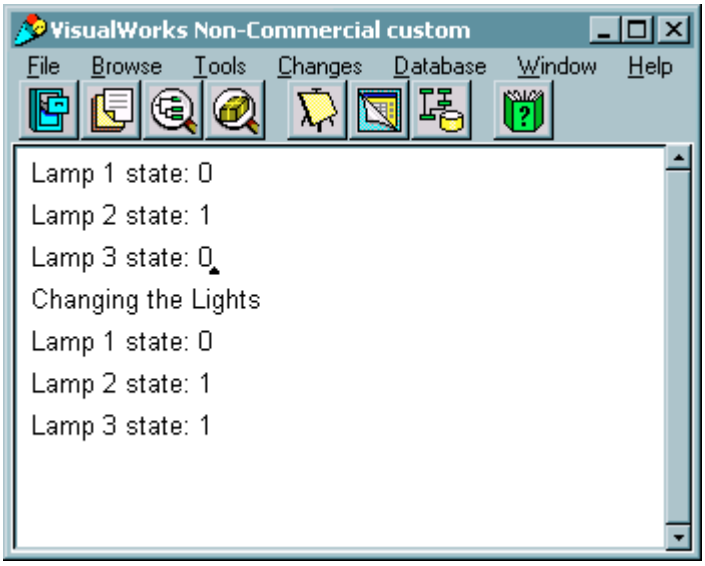
- Now we can call the method `showStates` to reveal the light that is on. Lamp 2 is initially set on, then the `changeLight` message is sent to `aTrafficLight`. Notice how Lamp 2 automatically turns itself off when `changeLight` turns on Lamp 3. This kind of automation is the primary value of the dependency mechanism.

```
| aTrafficLight |
        aTrafficLight := TrafficLight new.
        ((aTrafficLight lamplist) at: 2 ) state: 1.
        aTrafficLight showStates.
        aTrafficLight changeLight.
        aTrafficLight showStates.
```

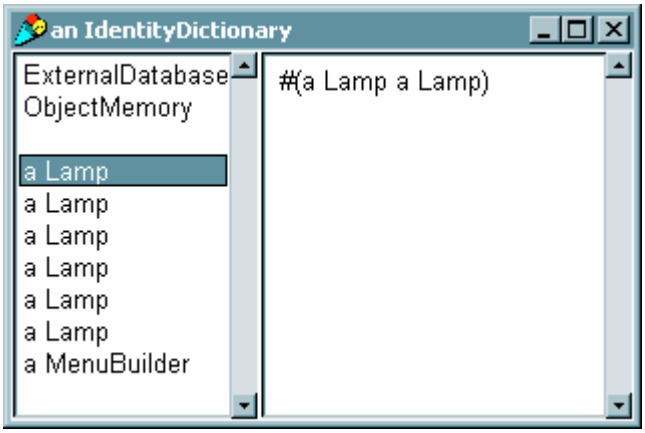


- What happens if we break the dependency on one of the lamp? Let us remove the 2nd lamp's dependency on 3rd lamp from the ordered collection. Notice that when the 3rd lamp turns on the 2nd lamp never turns off.

```
| aTrafficLight |
aTrafficLight := TrafficLight new.
((aTrafficLight lamplist) at: 2 ) state: 1.
aTrafficLight showStates.
(aTrafficLight lamplist at: 3) removeDependent:
      (aTrafficLight lamplist at: 2).
aTrafficLight changeLight.
aTrafficLight showStates.
```



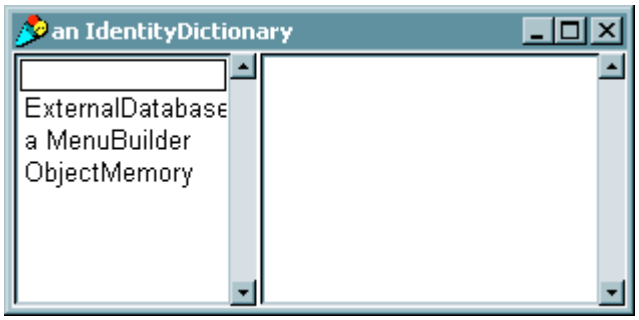
- For an arbitrary object, Smalltalk does not remove the dependents from the object when it is no longer in use.
- The system wide dependencies are stored in an identity dictionary `DependentsFields`. Upon inspection of this dictionary, we can see the dependencies we have created in the last two examples. Each lamp has an entry that includes each of the other lamps in the traffic light. Note that there are six lamps, as the two examples each added 3 lamps.



- Because we do not have a direct way to access the lamp objects from the previous examples, we will remove their dependencies by removing their entries in

`DependentsFields`. The following code will remove all keys of the object class Lamp. The code simply creates a collection of keys to be removed, then removes each one.

```
| keys |
keys := OrderedCollection new.
DependentsFields associationsDo: [ :anObject |
      ((anObject key) isKindOf: Lamp)
            ifTrue: [ keys add: (anObject key) ]].
              keys do: [ :aKey |
              Transcript show: 'Removing ', aKey printString; cr.
              DependentsFields removeKey: aKey ifAbsent: []].
```



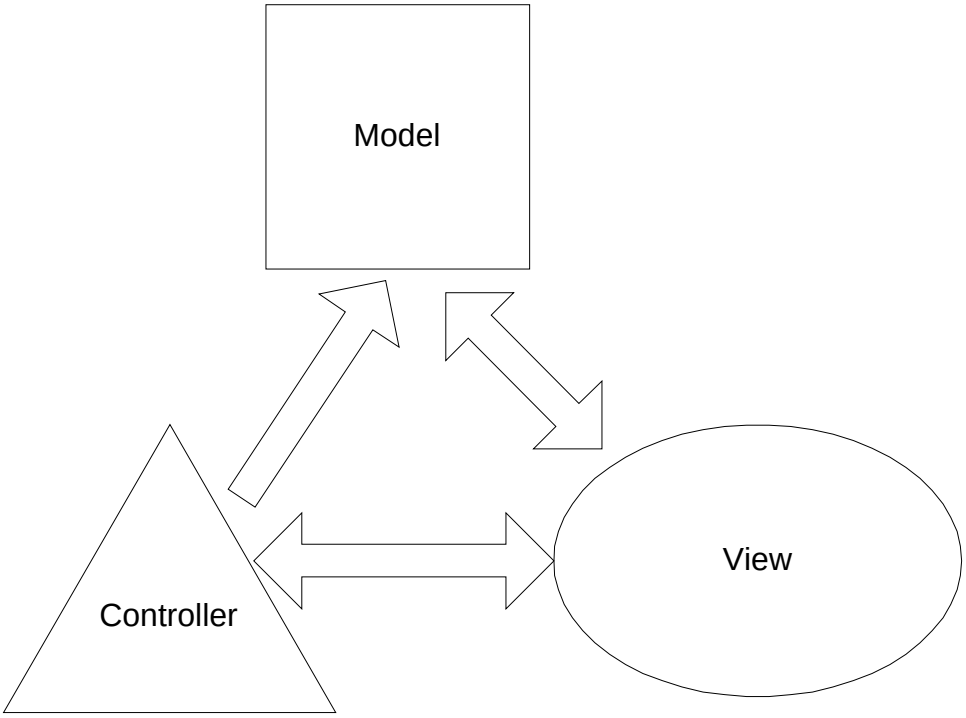
- To avoid leaving stray dependencies in `DependentsFields`, we need to add a method to `TrafficLight` that will remove the dependencies of the Lamps. This is done by simply sending the release message to each Lamp.

```
removeDependents
      "Removes the dependents of each lamp in the TrafficLight"

      lamplist do: [:lamp | lamp release ].
```

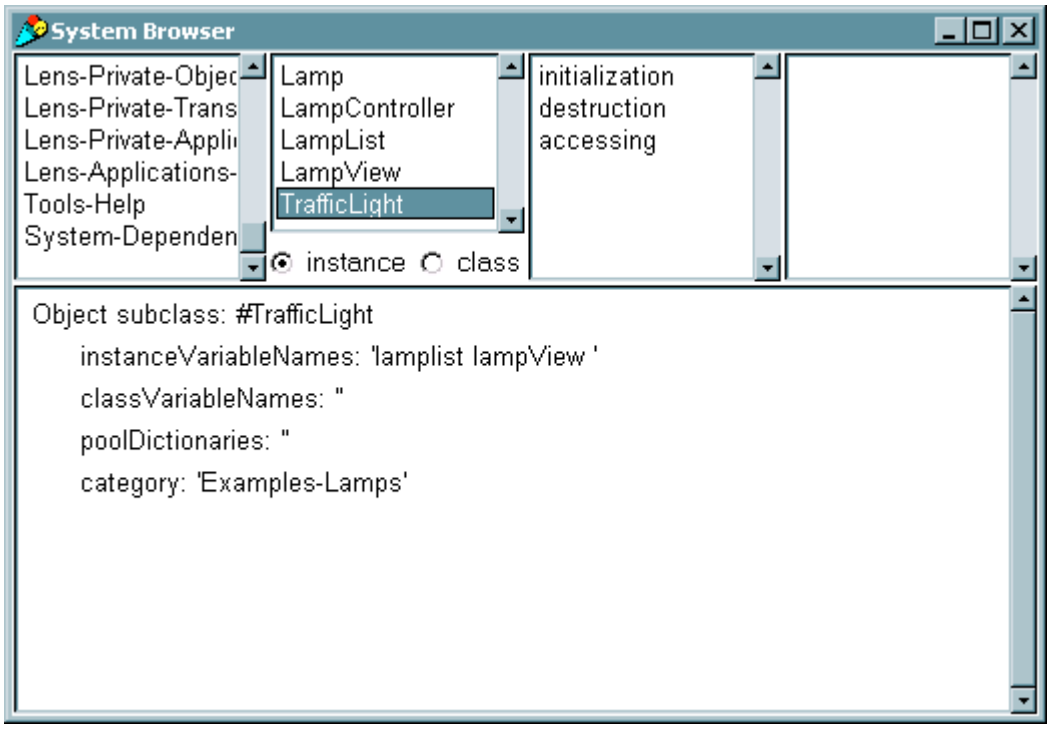
- After the last `showStates` message, we should now add
  `aTrafficLight removeDependents`.
- Inspection of `DepedentsFields` shows that we have removed all of the dependencies.

Lecture 21: The Model-View-Controller Paradigm

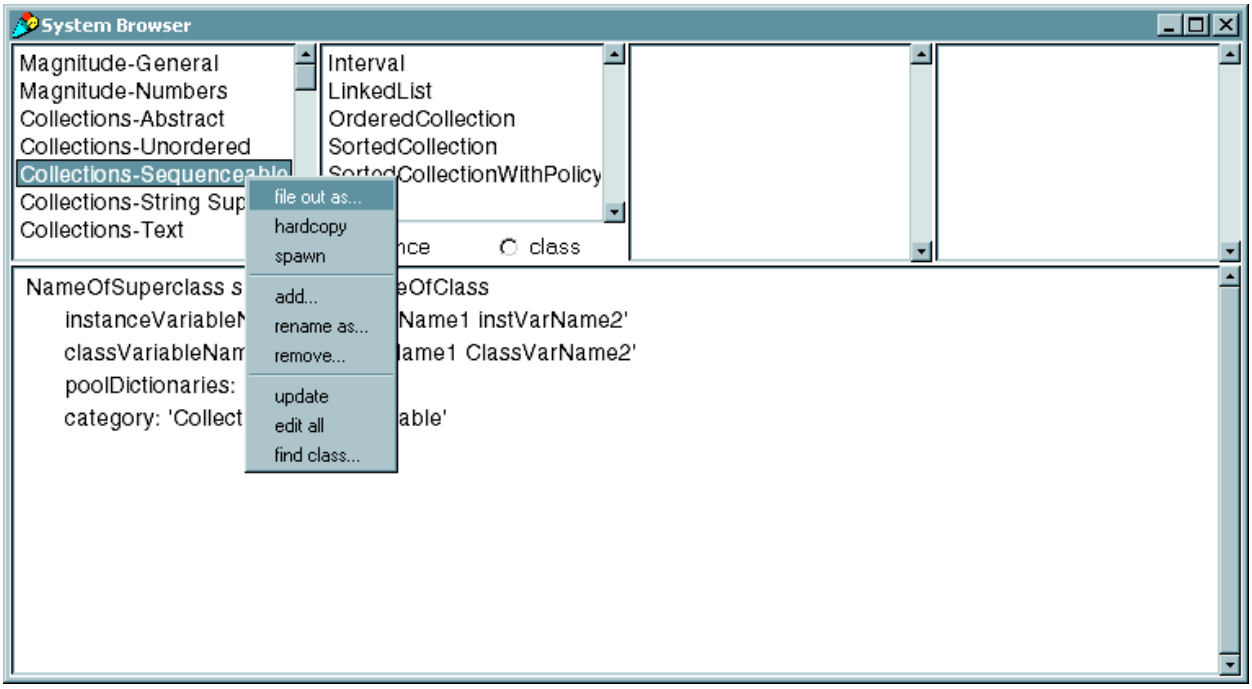
- **Definitions**
  - Model: The object to be looked at and/or modified
    - Provides the details to be displayed
  - View: The object that determines the precise manner in which the model is to be displayed (i.e. a window manager)
    - Displays the model and provides visual feedback for controller interactions
  - Controller: The object that handles the keyboard and mouse interactions for this view
  - The MVC Triad

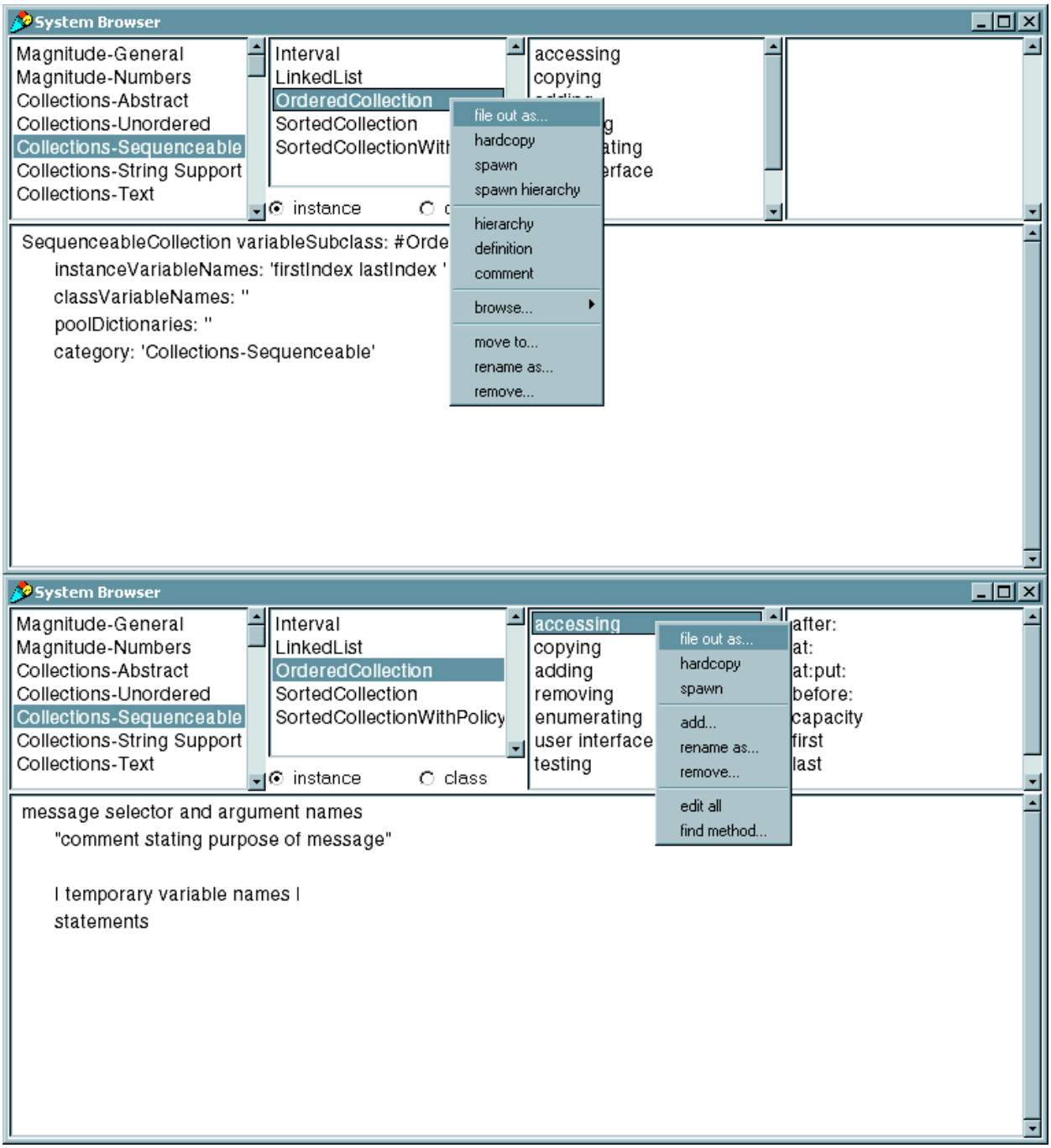


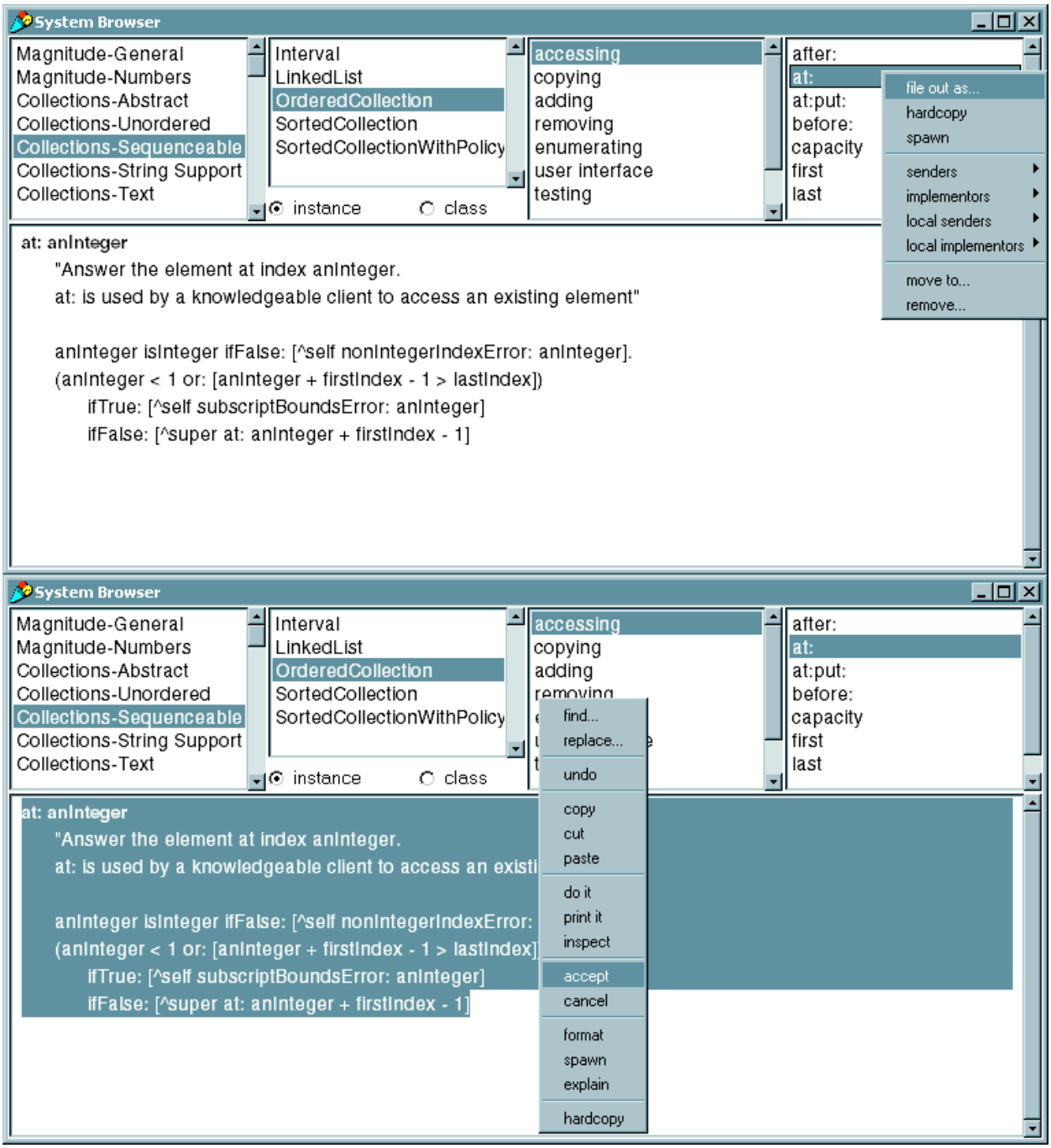
- The view and the controller interact to provide a graphical interface to the model.
  - An example of a MVC application is the browser. The browser is composed of 5 Views, each with its own controllers. The model contains the entire source, and the views and controllers interact to display the source code.



- Each panel of the browser has its own controller, notice how a right mouse button's menu is different in each panel. Its also important to note that each time a item in the SelectionView is clicked one, the other views change as well.




System Browser
Magnitude-General
Magnitude-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-String Support
Collections-Text
Interval
LinkedList
OrderedCollection
SortedCollection
SortedCollectionWith
instance
accessing
copying
file out as...
hardcopy
spawn
spawn hierarchy
hierarchy
definition
comment
browse...
move to...
rename as...
remove...
SequenceableCollection variableSubclass: #Orde
instanceVariableNames: 'firstIndex lastIndex '
classVariableNames: ''
poolDictionaries: ''
category: 'Collections-Sequenceable'
System Browser
Magnitude-General
Magnitude-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-String Support
Collections-Text
Interval
LinkedList
OrderedCollection
SortedCollection
SortedCollectionWithPolicy
instance
class
accessing
copying
adding
removing
enumerating
user interface
testing
file out as...
hardcopy
spawn
add...
rename as...
remove...
edit all
find method...
after:
at:
at:put:
before:
capacity
first
last
message selector and argument names
"comment stating purpose of message"
| temporary variable names |
statements


System Browser
Magnitude-General
Magnitude-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-String Support
Collections-Text
Interval
LinkedList
OrderedCollection
SortedCollection
SortedCollectionWithPolicy
instance
class
accessing
copying
adding
removing
enumerating
user interface
testing
after:
at:
at:put:
before:
capacity
first
last
file out as...
hardcopy
spawn
senders
implementors
local senders
local implementors
move to...
remove...
at: anInteger
"Answer the element at index anInteger.
at: is used by a knowledgeable client to access an existing element"
anInteger isInteger ifFalse: [^self nonIntegerIndexError: anInteger].
(anInteger < 1 or: [anInteger + firstIndex - 1 > lastIndex])
ifTrue: [^self subscriptBoundsError: anInteger]
ifFalse: [^super at: anInteger + firstIndex - 1]
System Browser
Magnitude-General
Magnitude-Numbers
Collections-Abstract
Collections-Unordered
Collections-Sequenceable
Collections-String Support
Collections-Text
Interval
LinkedList
OrderedCollection
SortedCollection
SortedCollectionWithPolicy
instance
class
accessing
copying
adding
after:
at:
at:put:
before:
capacity
first
last
find...
replace...
undo
copy
cut
paste
do it
print it
inspect
accept
cancel
format
spawn
explain
hardcopy
at: anInteger
"Answer the element at index anInteger.
at: is used by a knowledgeable client to access an existi
anInteger isInteger ifFalse: [^self nonIntegerIndexError:
(anInteger < 1 or: [anInteger + firstIndex - 1 > lastIndex]
ifTrue: [^self subscriptBoundsError: anInteger]
ifFalse: [^super at: anInteger + firstIndex - 1]

- **Model**
  - While the Object class handles dependency coordination, as seen in the TrafficLight/LampList example, most model objects are created as a subclass of Model.
  - Object vs. Model
- Object uses a global dictionary to store dependents.
  - This approach provides global dependency coordination, but dependents must be explicitly removed.
- Model holds the collection of dependents in an instance variable
  - The model is able to find the dependents faster, hence the methods involving the dependents is speeded up.
  - Failure to release an object can be safely ignored. Garbage collection is able to remove obsolete dependents.
  - The traffic light example presented in the previous lectures is an excellent example. To simplify the model, we will now focus on the `LampList` and `Lamp` classes.
  - To gain the features of Model, the `Lamp` and `LampList` classes should now be subclassed off `Model`, rather than `Object` or `OrderedCollection`. It is important to note that two instance variables have been added to `LampList`: `numin` and `theList`.
- `numin` is an internal counter, which will be discussed when its implementation is shown.
- `theList` is an `OrderedCollection` used to store the list of Lamps, since `LampList` is no longer subclassed off of `OrderedCollection`.

```
Model subclass: #Lamp
       instanceVariableNames: 'state id '
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Examples-Lamps'

Model variableSubclass: #LampList
       instanceVariableNames: 'numin list'
       classVariableNames: ''
       poolDictionaries: ''
       category: 'Examples-Lamps'
```

- Several changes must be made to existing methods, and new methods must be added to compensate for `LampList` no longer being subclassed off of `OrderedCollection`.
  - Initialization must now create `theList` as an `OrderedCollection`.

```
initialize
       "Sets the count to zero."
       theList := OrderedCollection new.
       numin := 0.
```

  - Now that LampList is subclassed off of Model, it is part of the dependency mechanism. To maintain the dependency updating process, LampList must implement the update: method. The instance variable numin is used in this method to count the number of Lamps that have reported in to the LampList object. Once all lamps have reported in, the LampList object can send the changed message.
  - In the next lecture on Views, we use the lamplist as a model for a LampView. We wait for all of the lamps to report in to avoid a race condition where the view is redrawn before all lamps have had a chance to update their state.

```
   update: signal
      "Waits for all lamps to report in, then redraws the view."

      numin := numin + 1.
      (numin = self size) ifTrue: [
             numin := 0.
             self changed.]
```

- Methods to access and add to the LampList must now include implementation of `add:`, `at:`, `do:`, `detect:`, and `size` so other methods will not break. It should be noticed, if other methods were needed, they could be implemented simply by passing the message to `theList`.

```
add: aLamp
       "Adds aLamp to theList."
       theList add: aLamp.
       ^theList.

at: anInteger
       "returns a Lamp for theList."
       ^(theList at: anInteger).

detect: aBlock
       "Passes a detect message to theList"
       ^(theList detect: aBlock).

do: aBlock
       "Tells theList to do aBlock."
       ^(theList do: aBlock).

size
       "Returns the size of theList."
       ^(theList size).
```

  - Rather than having the `TrafficLight` create the dependency, the method `make:` now adds each lamp to the `LampList` as a dependent, as well as make each lamp dependent on every other lamp.

```
make: anInteger
       "Makes a lamp list with anInteger number of
       lamps input by the user."

       | lamplist lamp |

       lamplist := LampList new: anInteger.
       anInteger
              timesRepeat:
                     [ lamp := Lamp new.
                       lamp id: (lamplist size + 1).
                       lamp state: 0.
                       lamplist add: lamp.
                       lamp addDependent: lamplist].
       1 to: lamplist size do: [ :l |
              1 to: lamplist size do: [ :dep |
                     l = dep ifFalse: [
                            (lamplist at: l) addDependent:
                                   (lamplist at: dep)]]].
       ^lamplist
```

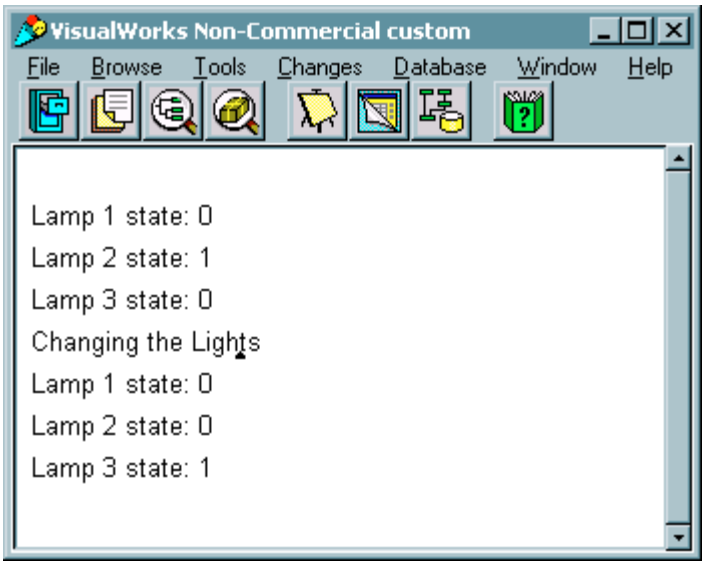
- No changes are needed for accommodating `Lamp's` new subclassing.
- With one modification, the examples used for the `TrafficLight` will work as well now. Since the `make:` method creates all of the dependencies, `TrafficLight's initialize` method only has to make the light.

```
initialize
   "Creates the three lights and turns the first on"

   lamplist := LampList make:3.
```

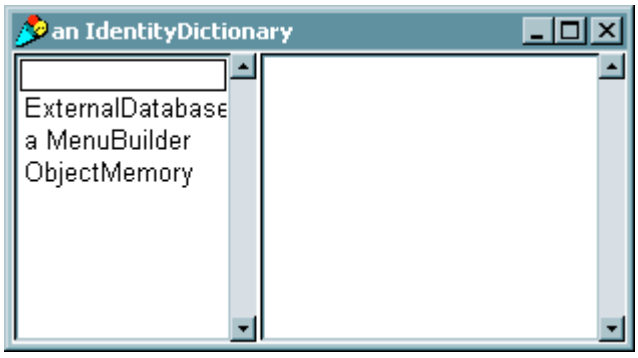
```
^self.
```

- Now we can look at the same code used in earlier `TrafficLight` examples.

```
| aTrafficLight |
      aTrafficLight := TrafficLight new.
      ((aTrafficLight lamplist) at: 2 ) state: 1.
      aTrafficLight showStates.
      aTrafficLight changeLight.
      aTrafficLight showStates.
      "Note – we no longer need to explicitly
      remove dependents"
```



- The exact same code used earlier produces the exact same output to the transcript. The only difference can be seen by inspecting `DependentsFields`. Notice that there are no dependents left behind? Since the `Lamp` and `LampList` objects were all subclassed off Model, the dependents were all stored locally in an instance variable, and removed once the object executed.



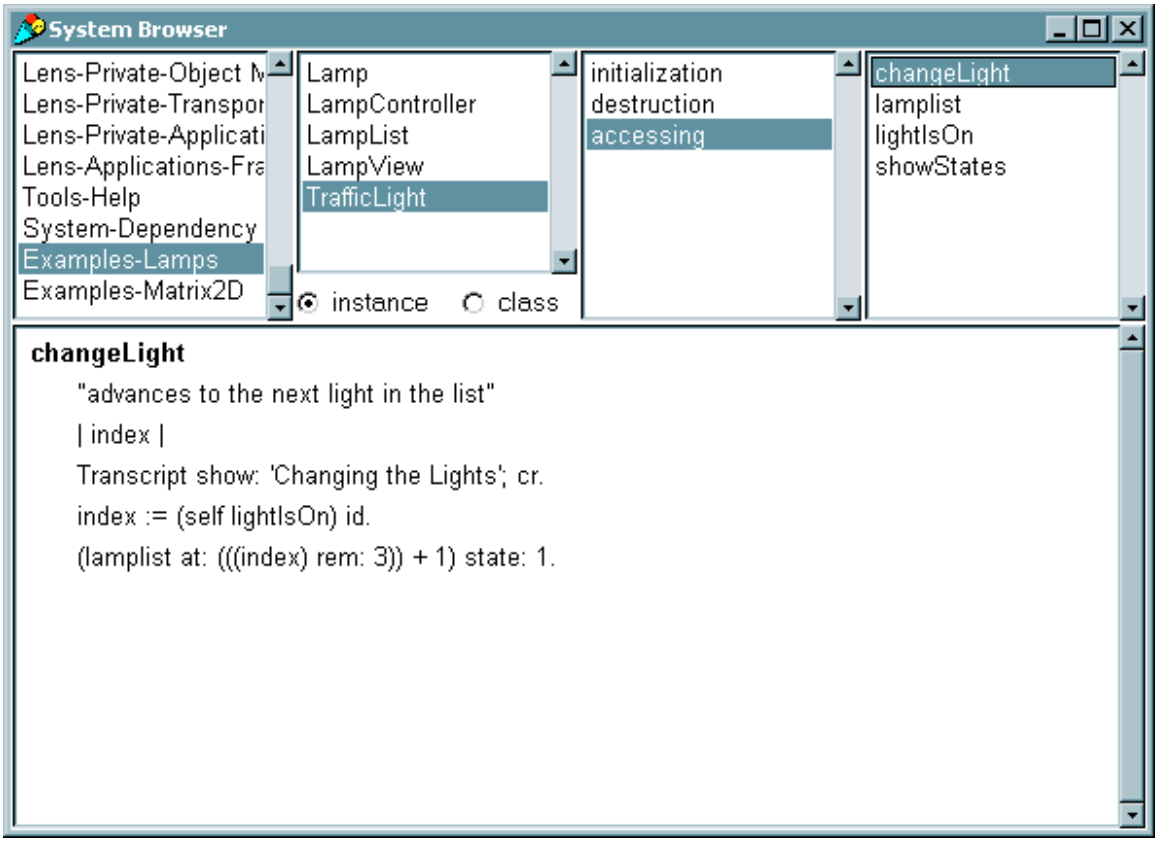
  - Given the simplicity of maintaining dependents, the model concept is used extensively in the following:
- Dependent Views, which ask for data

    - Example: Real-time graphs and charts, one model could feed two different windows data to be displayed.
- Controllers associated with dependent views, which supply user input data and request menu operations
    - Example: modifying the menu choices so the choices are different depending on what was clicked on
- Dependent buttons, which request button operations
    - Example: Grey'ing out inactive buttons
- Other models, which request data processing and other services
- The model itself, which requests data processing and other services.
- We will look at coupling the Model with a View and Controller in the next lectures

.

Lecture 22: The View

- **View**
  - The view is responsible for displaying aspects of the model. Because there are many kinds of models, there are many kinds of views, ranging from very simple to incredibly complex.
  - A view can be thought of as a part of a window in which a visual object is displayed. The object can be passive, such as an image or text, or be an active object that updates itself according to changes in the model, such as a real time graph.
- The browser window is an excellent example.
  - Each pane is a view. The four top panes are SelectionView objects, and the bottom pane is a TextCollectorView object.



  - Every view must implement the following instance methods
- `displayOn: #anAspect`
  - Completely builds the contents of the view
  - Called when a view is first created and each time the entire view is redrawn (e.g. uncovered by another window)
- `update: #anAspect`
  - Called whenever the model changes (e.g. sends itself a `changed:` message)
  - Used to reconstruct all or portions of a view depending on how the model was changed (indicated by `#anAspect` symbol)
  - If desired, `#anApsect` can be ignored in either of these methods.

- Suppose we wish to create a view for the Traffic Light example, we would now implement these methods and a class definition for a new class, `LampView`. The view will be an instance of `AutoScrollingView` with three lamps in it.

- In the class definition, an instance variable must be kept so the view knows what window it is in.

```
AutoScrollingView subclass: #LampView
    instanceVariableNames: 'window '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Examples-Lamps'
```

- For now, update: needs to only re-display the view.

```
update: aModel
        "The receiver's model has changed.  Redisplay the
        receiver."

        self displayObject
```

- `#anAspect` will be ignored, so we create a method `displayObject`, which is called by `displayOn:`. displayObject displays the on/off images for each lamp, depending on its state, then displays each image in its graphicsContext instance variable. graphicsContext is an instance of GraphicsContext, a feature-rich class used for drawing into a display surface, such as a view. More can be learned about this class from VisualWork's online documentation.

```
displayOn: ignored
        "Display the lamps in a window."

        self displayObject

displayObject
        "Display the lamps in the window."

        | image lamp |
        "model is a LampList"
        1 to: model size do:
                [ :index | lamp := model at: index.
                        lamp state = 0
                          ifTrue:
                            [ image := lamp getLampOffImage ]
                          ifFalse:
                            [ image := lamp getLampOnImage ].

                self graphicsContext displayImage: image
                    at: lamp position.].
```

- In `displayObject`, the methods `getLampOffImage` and `getLampOnImage` are used. These methods must be added to the Lamp instance methods. In addition to these methods, we need to add a class method, `initialize`, to create the images. You need not be concerned with the code within, just be aware that the `initialize` method exists.

```
getLampOffImage
        "Returns the image of the lamp in off state."

        ^LampOffImage

getLampOnImage
        "Returns the image of the lamp in on state."
```

```
        ^LampOnImage

initialize
        "Initialize class with an image."

        | bitPattern |

        bitPattern := #[
                        2r00001111 2r11110000
                        2r00110000 2r00001100
                        2r01000000 2r00000010
                        2r10000000 2r00000001
                        2r10000000 2r00000001
                        2r10000000 2r00000001
                        2r10000000 2r00000001
                        2r10000000 2r00000001
                        2r10000000 2r00000001
                        2r01000000 2r00000010
                        2r00100000 2r00000100
                        2r00010000 2r00001000
                        2r00001000 2r00010000
                        2r00000100 2r00100000
                        2r00000100 2r00100000
                        2r00000010 2r01000000
                        2r00000010 2r01000000
                        2r00000010 2r01000000
                        2r00000010 2r01000000
                        2r00000010 2r01000000 ].

        LampOnImage := Image
                extent: 16@20
                depth: 1
                palette: MappedPalette blackWhite
                bits: bitPattern
                pad: 8.

                LampOffImage := Image
                extent: 16@20
                depth: 1
                palette: MappedPalette whiteBlack
                bits: bitPattern
                pad: 8.
```

- `displayObject` still will not work properly. It references the position of each lamp, but until now the position has not been set. The position needs to be set somewhere, so we will set the position of each lamp in `LampList's` make method.

```
make: anInteger
        "Makes a lamp list with anInteger number of lamps input by
the user."

        | lamplist lamp |
        lamplist := LampList new: anInteger.
        anInteger
                timesRepeat:
                        [ lamp := Lamp new.
                          lamp position: 25 @ (lamplist size * 30).
                          lamp id: (lamplist size + 1).
                          lamp state: 0.
                          lamplist add: lamp.
                          lamp addDependent: lamplist].
```

```
        1 to: lamplist size do: [ :l |
                1 to: lamplist size do: [ :dep |
                        l = dep ifFalse: [
                                (lamplist at: l) addDependent:
(lamplist at: dep)]]].
        ^lamplist
```

- Now we have the methods to create a visual representation of a lamp, and the methods to update the view, but we still need to attach the model to the view and create the window to put the view in. To be displayed on the screen, a view must be contained in an instance of `ScheduledWindow`.
  - Registering the view as a dependent of the model is simple through the use of the message `model: aModel`
- `ScheduledWindow` has a model and a controller
  - The "Scheduled" part of `ScheduledWindow` refers to the fact that `ScheduledWindow` is part of `ScheduledController`, the control manager.
- Usually, the `ScheduledWindow` is created, and its visual components are added before it is opened. The following code demonstrates this:

```
| aWindow |
aWindow := ScheduledWindow new.
aWindow
        component: 'Hello World' asComposedText.
aWindow openIn: (20 @ 20 extent: 150 @ 150 ).
```

  - Now we can create a new view, place it inside a window and register the model in the same class method for `LampView`:

```
openOn: aLampList
        "Creates a new Lamp View on aLampList."

        | view window |
        view := self new.

        "Register the model"
        view model: aLampList.

        window := ScheduledWindow new.
        window label: 'Lamp Viewer'.
        window minimumSize: 50@100.
        window insideColor:
                (ColorValue red: 1.0 green: 0.0 blue: 0.0 ).
        window component: view.
        view window: window.
        window open.
        ^view.
```

- The last task that must be done is the simplest: create an instance of `LampView` in the `TrafficLight`. The simplest way is by modifying the `initialize` method:

```
initialize
    "Creates the three lights and turns the first on"

    lamplist := LampList make:3.
    lampView := LampView openOn: lamplist.
    ^self.
```

- The following code will create the light, initialize it to the second light, then changes lights every second for 10 seconds. The screen capture is the resulting window after 11 seconds (black was used as “on”).

```
| aTrafficLight |
aTrafficLight := TrafficLight new.
((aTrafficLight lamplist) at: 2 ) state: 1.
10 timesRepeat:
    [(Delay forSeconds: 1) wait.
    aTrafficLight changeLight].
```

Lecture 23: The Controller

- Now, suppose we want to use the code to create a control panel with Lamps, rather than a traffic light. We also want to control the lamps without entering commands into the workspace. We need to modify the controller.
  - Controllers serve two primary purposes, event handling and menu pop-ups. For now, we will focus on event handling.
- Every controller has a `controlActivity` method which functions as an event handler. The `controlActivity` method is repeatedly invoked while control is active (e.g. the mouse pointer is in the view of a window). It is in this method that we check for events from the user, such as key presses and mouse button pushes, by sending messages to a sensor.
- Each window has an input sensor, an instance of WindowSensor. The sensor holds queues for keyboard events and window sizing/moving/closing events. It also knows the state of the mouse, including the position of the pointer and the states of the buttons.
- Lets start constructing the `controlActivity` method for the `LampController` by first checking for keyboard input. We'll use the number keys, 1, 2, and 3, to turn on the corresponding lamp. First we check to see if a key was pressed by using the `keyboardPressed` message:

```
sensor keyboardPressed.
```

  - If this message returns true, then a key has been pressed and we need to determine which one. The sensor will return the character pressed when we send it the `keyboard` message. We then convert the resulting character to an integer and test if it is a valid lamp number. If so, we set the state of the lamp to "on".
  - In the Smalltalk tradition, we want to keep the `controlActivity` method short, so we'll put the keyboard processing code in a separate method.

```
controlActivity
    "Do this when the mouse is in the window."

    (sensor keyboardPressed)
           ifTrue: [ self processKeyboard]

processKeyboard

    | int |
    int := sensor keyboard digitValue.
    (int between: 1 and: model size)
           ifTrue: [(model at: int) state: 1].
```

- We want to add some way to quit the application, but request confirmation when the user chooses to quit. Lets implements this when the user presses the yellow (middle for 3 button mice, right for 2 button mice) mouse button. We can detect a mouse button by sending one of the following messages to the sensor:
  - `redButtonPressed`
  - `yellowButtonPressed`
  - `blueButtonPressed`.
- In our case, we use `sensor yellowButtonPressed`. If this method returns true, then the `confirm:` message is sent to the Dialog class to bring up a window with "yes" and "no" buttons. Subsequent mouse presses are ignored until one of the confirm buttons is pressed. If the "yes" button is pressed, the confirm message returns true and the window is closed.
- Below we implement the yellow button activity method that is called when a yellow button press is detected in the `controlActivity` method.

```
controlActivity
	"Do this when the mouse is in the window."

	(sensor keyboardPressed)
		ifTrue: [ self processKeyboard]
		ifFalse: [
			sensor yellowButtonPressed
			ifTrue: [ self processYellowButton].
		].

processYellowButton
	“This method is called when the yellow button
	is pressed”

	(Dialog confirm: 'Quit ?')
		ifTrue: [
		  view window controller closeAndUnschedule].
```

- Lastly, it would be convenient if each lamp would turn on (and turn off all other lamps) by simply clicking on it. We will use the red (left mouse button) for this operation. As with the yellow button, we detect the red button press in the `controlActivity` method `sensor redButtonPressed`, then send the `processRedButton` message if the result is true.
- To determine if a lamp was clicked on, we compute a Rectangle (in view coordinates) which bounds the lamp image. Then we check to see if the point where the red button was clicked is contained in the bounding rectangle.
- We iterate through the LampList until we find a lamp that has been clicked on, in which case we change that lamp’s state to #on, or until we have examined all lamps.
- The following code implements the algorithm discussed above:

```
processRedButton
   "This method is called when the red button is pressed."

   | mpt image box |

   "Wait for the mouse button to be released."
   sensor waitNoButton.

   "Get the point where the red mouse button was last
            pressed down."
   mpt := sensor lastDownPoint.

   "Assuming all lamp images are the same, get the
            first lamp image to use for computation"
   image := (model at: 1) getLampOffImage.

      "Now iterate through each lamp in the model or
            until we find one that has been clicked on."
   1 to: (model size) do: [ :lampNumber | | lamp |
          lamp := (model at: lampNumber).

             "Compute the bounding box of this lamp’s image
                    in the view’s coordinates."
          box := Rectangle origin: (lamp position)
                    extent: (image extent).

          "Check if the pointer was on the image when
                the button was pressed."
          (box containsPoint: mpt) ifTrue: [
                "If so, then turn that lamp on."
                ^lamp state: 1].
```

```
        ].
```

- Finally, included for completeness is the class definition as well as the complete `controlActivity` method.

```
Controller subclass: #LampController
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples-Lamps'

controlActivity
   "Do this when the mouse is in the window."

   (sensor keyboardPressed)
          ifTrue: [ self processKeyboard]
          ifFalse: [
                 sensor yellowButtonPressed
                 ifTrue: [ self processYellowButton].
                 sensor redButtonPressed
                 ifTrue: [ self processRedButton].
```

Appendix1: VisualWorks 2.5 versus Smalltalk-80

	VisualWorks 2.5	Smalltalk-80
Assignment	:=	←
Global Variables	Start with Caps	Does not care
DeepCopy	Removed	Present
Fractions	asRational	asFraction
FileName protocol	fileNamed:	named:

Differences found throughout the lecture note's examples

- **Classes removed from VisualWorks 2.5**
  - Button
  - DebuggerController
  - DebuggerTextView
  - DialogCompositController
  - DialogController
  - DialogView
  - FixedThumbScrollbar
  - FractionalWidgetView
  - HandlerController
  - ListController
  - ListView
  - SelectionSetInListController
  - SelectionInListView
  - TextItemView
  - TextItemEditor
  - TextController
  - TextView
  - PopUpMenu
  - WidgetSpecification

Appendix2: VisualWorks rules and Smalltalk Syntax

- **Capitalization rules**
  - Upper Case
    - Class names
    - Class variables and global variables
  - Lower Case
    - Method names
    - Temp variables, instance variables, class instance variables, method arguments
  - Use embedded capital letters, not underscores
- **Reserved words**
  - `nil`
  - `true`
  - `false`
  - `self`
  - `super`
  - `thisContext`
- **Operators**
  - `:=`
    - called ‘gets’ operator, used for assignment
  - `^`
    - called ‘returns’ operator, used to return a value
  - Example

```
name: aSymbol
        name := aSymbol.
        ^name.
```

- **Literals**
  - use VisualWorks syntax chapter for reference here
  - Numbers
  - Characters
  - Strings
  - Symbols
  - Arrays of literals
  - Byte Arrays (notice the use of brackets)
- **Comments**
  - “Comment”
  - periods allowed within double quotes

Appendix 3: A List of Methods for the System Classes

Magnitude:

Creation:

Operations:

< aMagnitude

less than operator returns boolean

<= aMagnitude

less than or equal operator returns boolean

> aMagnitude

greater than operator returns boolean

>= aMagnitude

greater than or equal operator returns boolean

between: min and: max

returns True if object's magnitude is between min and max

min: aMagnitude

returns the lesser of the object and aMagnitude

max: aMagnitude

returns the greater of the object and aMagnitude

Magnitude->Date:

Creation:

today

instance representing the current date

fromDays: dayCount

instance representing the date dayCount days from 01/01/1901

newDay: day month: monthName year: yearInteger

instance representing day number of days into monthName in yearInteger

newDay: dayCount year: yearInteger

instance representing dayCount days into yearInteger

Operations:

dayOfWeek: dayName

returns index of dayName in the week, #Sunday = 0

nameOfDay: dayIndex

returns Symbol representing the day whose index is dayIndex

indexOfMonth: monthName

returns index of monthName in the year, #January = 0

nameOfMonth: monthIndex

returns Symbol representing the month whose index is monthIndex

daysInMonth: monthName forYear: yearInteger

returns Integer representing the number of days in monthName for year yearInteger

daysInYear: yearInteger

returns Integer representing the thumber of days in yearInteger

leapYear: yearInteger

returns 1 if yearInteger is a leap year, 0 otherwise

dateAndTimeNow

returns Array whose first element is current date, and whose second element is the current time

addDays: dayCount

returns Date that is dayCount days after object

subtractDays: dayCount

returns Date that is dayCount days before object

subtractDate: aDate

asSeconds
returns number of seconds between a time on 01/01/1901 and the same time in the receiver's day

Magnitude->Time:
Creation:
now
instance representing the current time
fromSeconds: secondCount
instance representing the time of secondCount after midnight
Operations:
millisecondClockValue
returns number of milliseconds since the millisecond clock was reset or rolled over
millisecondsToRun: timedBlock
returns number of milliseconds timedBlock takes to execute
timeWords
returns the number of seconds since 01/01/1901 (GMT) in 4 element byte array
totalSeconds
returns total number of seconds since 01/01/1901, correcting the time zone and daylight savings
dateAndTimeNow
returns Array whose first element is current date, and whose second element is the current time
addTime: timeAmount
returns Time that is timeAmount days after receiver
subtractTime: timeAmount
returns Date that is timeAmount before receiver
asSeconds
returns number of seconds since midnight that receiver represents

Magnitude->Character:
Creation:
value: anInteger
instance of Character which is the ASCII representation of anInteger
digitValue: anInteger
instance of Character which is the character representation of a munber of radix 35- $0 returns 0, $A returns 10, $Z returns 35
Operations:
asciiValue
returns Integer of ascii character
digitValue
returns Integer representing numerical radix
isAlphaNumeric
true if receiver is letter or digit
isDigit
true if receiver is digit
isLetter
true if receiver is letter
isLowercase
true if receiver is lowercase
isUppercase
true if receiver is uppercase
isSeparator
true if receiver is space, tab, cr, line feed, or form feed
isVowel

true if receiver is a,e,i,o,u

Magnitude->Number:
Creation:
Operations:
+ aNumber
returns sum of receiver and aNumber
- aNumber
returns difference of reciever and aNumber
* aNumber
returns result of mulitplying receiver by aNumber
/ aNumber
returns result of dividing receiver by aNumber. If result is not a whole number, then an instance of Fraction is returned
// aNumber
returns Integer result of division truncated toward negative infinity
\\ aNumber
returns Integer representing receiver modulus aNumber
abs
returns Number representing absolute value of receiver
negated
returns Number representing additive reciprocal
quo: aNumber
returns quotient of receiver divided by aNumber
rem: aNumber
returns remainder of receiver divided by aNumber
reciprocal
returns multiplicative reciprocal (1/reciever)
exp
returns e raised to the power of receiver
ln
returns natual log of receiver
log: aNumber
returns log base aNumber of receiver
floorLog: radix
returns floor of log base radix of reciever
raisedTo: aNumber
returns result of raising receiver to aNumber
raisedToInteger: anInteger
returns result of raising receiver to anInteger, where anInteger must be an Integer
sqrt
returns square root of receiver
squared
returns receiver raised to the second power
even
true if receiver is even
odd
true if receiver is odd
negative
true if receiver is <= 0
positive
true if receiver is >= 0
strictlyPositive
true if receiver > 0
sign

returns 1 if receiver > 0, 0 if receiver == 0. –1 if receiver < 0

ceiling
returns result of rounding towards positive infinity

floor
returns result of rounding towards negative infinity

truncated
returns result of rounding towards zero

truncateTo: aNumber
returns result of truncating to multiple of aNumber

rounded
returns result of rounding receiver

roundedTo: aNumber
returns result of rounding receiver to neastest multiple of aNumber

degreesToRadians
returns Float of radian representation of receiver. Assumes receiver is in degrees

radiansToDegrees
returns Float in degrees of conversion of receiver. Assumes receiver is in radians

sin
returns Float of sin(receiver) in radians

cos
returns Float of cos(receiver) in radians

tan
returns Float of tan(receiver) in radians

arcSin
returns Float of arcSin(receiver) in radians

arcCos
returns Float of arcCos(receiver) in radians

arcTan
returns Float of arcTan(receiver) in radians

coerce: aNumber
casts receiver as same type as aNumber

generality
returns the number respresenting the ordering of the receiver in the generality heirarchy

retry: aSymbol coercing: aNumber
an arithmetic operation aSymbol could not be performed, so the operation is retried casting the receiver or argument to aNumber (picking the lowest order of generality)

Magnitude->Number->Integer:

Creation:

Operations:

factorial
returns Integer representing the factorial of the receiver

gcd: anInteger
returns Integer representing the Greatest Common Denominator of the receiver and anInteger

lcm: anInteger
returns Integer representing the Lowest Common Multiple of the receiver and anInteger

allMask: anInteger
treat anInteger as a bit mask. Returns True if all 1’s in anInteger are 1 in the receiver

anyMask: anInteger
treat anInteger as a bit mask. Returns True if any on the 1’s in anInteger are 1 in the receiver

noMask: anInteger
treat an Integer as a bit mask. Returns True if none of the 1's in anInteger are 1 in the receiver
bitAnd: anInteger
returns Integer representing a boolean AND operation between anInteger and the receiver
bitOr: anInteger
returns Integer representing a boolean OR operation between anInteger and the receiver
bitXor: anInteger
returns Integer representing a boolean XOR (eXclusive OR) operation between anInteger and the receiver
bitAt: anIndex
returns the bit (0 or 1) at anIndex
bitInvert
returns an Integer which is the complement of the receiver
highBit
returns an Integer representing the index of the highest order bit
bitShift: anInteger
returns an Integer whose value (in two's-complement) is the receiver's value shifted anInteger number of bits. Negative shifts are to the right.

Random

Creation:

:= Random new
instance representation of a random number generator
next
instance of a random rumber. The receiver must be a random number generator, which has previously been started

Operations:

Collection

Creation:

#(Object1, Object2, Object3, Object4)
instance representing an array containing up to 4 objects passed as arguments
new
instance representing an empty collection
new:
instance representing a collection
with: anObject
instance representing a collection containing anObject
with: firstObject with: secondObject
instance representing a collection containing firstObject and secondObject

Operations:

add: newObject
adds newObject to the receiver and returns newObject
addAll: aCollection
adds aCollection to the receiver and returns aCollection
remove: oldObject
removes oldObject from the receiver and returns oldObject unless there is no object oldObject (reports an error).
remove: oldObject ifAbsent: anExceptionBlock
removes oldObject from the receiver, unless it does not exist, in which case anExceptionBlock is executed. Returns oldObject or result of anExceptionBlock
removeAll: aCollection

removes all elements of aCollection from the receiver and returns aCollection, unless not all elements of aCollection were present in the receiver, in which case an error is reported.

includes: anObject
returns True if anObject is an element of the receiver

isEmpty
returns True if the receiver has no elements

occurrencesOf: anObject
returns an Integer representing the number of occurrences of anObject in the receiver

do: aBlock
evaluate aBlock for every element of the receiver

select: aBlock
evaluates aBlock for every element of the receiver. Returns a new Collection containing all elements of the receiver for which aBlock evaluated to true

reject: aBlock
evaluates aBlock for every element of the reciever. Returns a new Collection containing all elements for which aBlock evaluated to false

collect: aBlock
evaluates aBlock for every element of the receiver. Returns a new Collection containing the results of every evaluation of aBlock.

detect: aBlock
evaluates aBlock for every element of the receiver. Returns the object which is the first element in the receiver for which aBlock evaluated to true. If no object evaluated to true, an error is reported.

detect: aBlock ifNone: exceptionBlock
evaluates aBlock for every element of the receiver. Returns the object which is the first element in the receiver for which aBlock evaluated to true. If no object evaluated to true, exceptionBlock is evaluated.

inject: thisValue into: binaryBlock
Evaluates binaryBlock for each element of the receiver, initializing a local variable to thisValue. Returns final value of the block. BinaryBlock has two arguments.

asBag
Returns a Bag with the elements from the receiver

asSet
Returns a Set with the elements from the receiver

asOrderedCollection
Returns an OrderedCollection with the elements from the reciever

asSortedCollection
Returns a SortedCollection with the elements from the receiver, sorted to each element is less than or eaqual to its successor

asSortedCollection: aBlock
Returns a SortedCollection with the elements from the receiver, sorted according to the arguemnt aBlock

Collection->Bag

Creation:

Operations:

add: newObject withOccureneces: anInteger
Adds anInteger number of occurrences of newObject to the receiver, and returns newObject

Collection->Set

Creation:

Operations:

Collection->Set->Dictionary and Collection->Set->IdentityDictionary

Creation:

Operations:

at: key ifAbsent: aBlock

Returns the value named by key. If the key is not present in the dictionary, returns evaluation of aBlock

associationAt: key

Returns the association named by key. If key is not present, an error is reported

associationAt: key ifAbsent: aBlock

Returns the association named by key. If key is not present, returns the evaluation of aBlock.

keyAtValue: value

Returns the name found first for value, or nil if value is not present

keyAtValue: value ifAbsent: exceptionBlock

Returns the name found first for value, or the evaluation of exceptionBlock if value is not found

keys

Returns Set representing all of the receiver's keys

values

Returns Set containing all of the receiver's values

includesAssociation: anAssociation

Returns true if anAssociation is included in the receiver

includesKey: key

Returns true if key is included in the receiver

removeAssociation: anAssociation

Removes anAssociation from the receiver. Returns anAssociation

removeKey: key

Removes key and associated value from the receiver. Returns value associated with key if key is included in the receiver, otherwise an error is reported

removeKey: key ifAbsent: aBlock

Removes key and associated value from the receiver. Returns value associated with the key if key is included in the receiver, otherwise returns the evaluation of aBlock

associationsDo: aBlock

Evaluate aBlock for each of the receiver's associations

keysDo: aBlock

Evaluate aBlock for each of the receiver's keys

Collection->SequenceableCollection

Creation:

Operations:

atAll: aCollection put: anObject

Associate each element of aCollection with anObject.

atAllPut: anObject

Put anObject as every one of the receiver's elements

first

Returns the first element of the receiver

last

Returns the last element of the receiver

indexOf: anElement

Returns an Integer representing the index of anElement in the receiver, 0 if not present

indexOf: anElement ifAbsent: exceptionBlock

Returns an Integer representing the index of anElement in the receiver, or the evaluation of exceptionBlock if anElement is not in the receiver

indexOfSubCollection: aSubCollection startingAt: anIndex
If the elements of aSubCollection appear in order in the receiver, returns the index of the first element of aSubCollection in the receiver, otherwise returns 0
indexOfSubCollection: aSubCollection: startingAt: anIndex ifAbsent: exceptionBlock
Returns the index of the first element of aSubCollection in the receiver if the elements of aSubCollection appear in order, otherwise returns the evaluation of aBlock
replaceFrom: start to: stop with: replacementCollection
Associates every element of the receiver from start to stop with the elements of replacementCollection and returns the receiver. The size of replacementCollection must equal start + stop + 1.
replaceFrom: start to: stop with: replacementCollection startingAt: repStart
Associates every element of the receiver from start to stop with the elements of replacementCollection starting with index repStart in replacementCollection. The receiver is returned
, aSequencableCollection
Returns the receiver concatenated with aSequencableCollection
copyFrom: start to: stop
Returns a subset of the receiver starting at index start and ending an index stop
copyReplaceAll: oldSubCollection with: newSubCollection
Returns a copy of the receiver with all occurrences of oldSubCollection replaced with newSubCollection
copyWith: newElement
Returns a copy of the receiver with newElement added on to the end
copyWithout: oldElement
Returns a copy of the receiver without all occurrences of oldElement
findFirst: aBlock
Evaluates aBlock for every element of the receiver and returns the index of the first element for which aBlock evaluates to true.
findLast: aBlock
Evaluates aBlock for each element of the receiver and returns the index of the last element for which aBlock evaluates to true
reverseDo: aBlock
Evaluates aBlock for each element of the receiver, starting with the last element
with: aSequenceableCollection do: aBlock
Evaluates aBlock for each element of the receiver and each element of aSequenceableCollection. The number of elements in aSequenceableCollection must equal the number of elements in the receiver and aBlock must have two arguements

Collection->SequenceableCollection->OrderedCollection
Creation:
Operations:
after: oldObject
Returns the element occurring after oldObject, or reports an error if oldObject is not found or is the last element
before: oldObject
Returns the element occurring before oldObject, or reports an error if oldObject is not found or is the first element
add: newObject after: oldObject
Inserts newObject after oldObject into the receiver and returns newObject unless oldObject is not found, in which case an error is reported
add: newObject before: oldObject
Inserts newObject before oldObject into the receiver and returns newObject unless oldObject is not found, in which case an error is reported
addAllFirst: anOrderedCollection

Adds each element of anOrderedCollection to the beginning of the receiver and returns anOrderedCollection

addAllLast: anOrderedCollection

Adds each element of anOrderedCollection to the end of the receiver and returns anOrderedCollection

addFirst: newObject

Adds newObject to the beginning of the receiver and returns newObject

addLast: newObject

Adds newObject to the end of the receiver and returns newObject

removeFirst

Removes the first object from the receiver and returns it, unless the receiver is empty in which case an error is reported

removeLast

Removes the last object from the receiver and returns it, unless the receiver is empty in which case an error is reported

Collection->SequenceableCollection->OrderedCollection->SortedCollection

Creation:

sortBlock: aBlock

Instance representing an empty SortedCollection using aBlock to sort its elements

Operations:

sortBlock

Returns the block that is to be used to sort the elements of the receiver

sortBlock: aBlock

Make aBlock the block used to sort the elements of the receiver

Collection->SequenceableCollection->LinkedList

Creation:

nextLink: aLink

Instance of Link that references aLink

Operations:

nextLink

Returns the receiver's reference

nextLink: aLink

Sets the receiver's reference to be aLink

addFirst: aLink

Adds aLink to the beginning of the receiver's list and returns aLink

addLast: aLink

Adds aLink to the end of the receiver's list and returns aLink

removeFirst

Removes the first element from the receiver's list and returns it. If the list is empty an error is reported

removeLast

Removes the last element from the receiver's list and returns it. If the list is empty an error is reported

Collection->SequenceableCollection->Interval

Creation:

from: startInteger to: stopInteger

Instance starting with the number startInteger and ending with stopInteger, incrementing by one

from: startInteger to: stopInteger by: stepInteger

Instance starting with the number startInteger and ending with stopInteger, incrementing by stepInteger

Operations:

Collection->SequenceableCollection->ArrayedCollection
Creation:
Operations:
Collection->SequenceableCollection->ArrayedCollection->CharacterArray->String
Creation:
Operations:
< aString
Returns true if the receiver collates before aString. Case is ignored.
<= aString
Returns true if the receiver collates before aString, or is the same as aString. Case is ignored.
> aString
Returns true if the receiver collates after aString. Case is ignored.
>= aString
Returns true if the receiver collates after aString, or is the same as aString. Case is ignored.
match: aString
Treats the receiver as a pattern containing #'s and *'s which are wild cards (# represents one character, * represents a substring). Returns true if the reciever matches aString. Case is ignored.
sameAs: aString
Returns true if the receiver collates exactly with aString. Case is ignored.
asLowercase
Returns a String representing the receiver in all lowercase
asUppercase
Returns a String representing the receiver in all uppercase
asSymbol
Returns a Symbol whose characters are the characters of the receiver

Collection->SequenceableCollection->ArrayedCollection->CharacterArray->Symbol
Creation:
intern: aString
Returns an instance of a Symbol whose characters are those of aString
internCharacter: aCharacter
Returns an instance of a Symbol which consists of aCharacter
Operations:

Collection->MappedCollection
Creation:
Operations:

Stream
Creation:
Operations:
next
Returns the next object accessible by the receiver
next: anInteger
Returns the next anInteger objects accessible by the receiver
nextMatchFor: anObject
Accesses the next object and returns true if it is equal to anObject
contents
Returns all of the objects in the collection accessed by the receiver.
nextPut: anObject

Stores anObject as the next object accessible by the receiver and returns anObject

nextPutAll: aCollection

Store the elements in aCollection as the next objects accessible by the receiver and returns aCollection. Advances the position reference to the new object.

next: anInteger put: anObject

Store anObject as the next anInteger number of objects accessible by the receiver and returns anObject. Advances the position reference to the new object.

atEnd

Returns true if the are no more objects accessible by the receiver

do: aBlock

Evaluate aBlock for each of the remaining objects accessible by the receiver

Stream->PositionableStream

Creation:

on: aCollection

Returns an instance which streams over aCollection

on: aCollection from: firstIndex to: lastIndex

Returns an instance which streams over a copy of a subcollection of aCollection from firstIndex to lastIndex

Operations:

isEmpty

Returns true if the collection the receiver accesses has no elements

peek

Returns the next object in the collection but does not increment the position reference

peekFor: anObject

Does a peek, if the next object is equal to anObject, then returns true and increments the position reference, otherwise just returns false

upTo: anObject

Returns a collection of the elements starting with the next object accessed by the receiver up to, but not including, anObject. If anObject is not an element of the remainder of the collection, then the entire remaining collection is returned.

position

Returns the receiver's current position reference

position: anInteger

Sets the receiver's position to anInteger. If anInteger exceeds the bounds of the collection, then an error is reported

reset

Sets the receiver's position to the beginning of the collection

setToEnd

Sets the receiver's position to the end of the collection

skip: anInteger

Sets the receiver's position to the current position + anInteger

skipThrough: anObject

Sets the receiver's position to be past ther next occurrence of anObject. Returns true if anObject occurs in the collection

Stream->PositionableStream->ReadStream

Creation:

Operations:

Stream->PositionableStream->WriteStream

Creation:

Operations:

cr
Stores the carriage return as the next element of the receiver
crtab
Stores the carriage return and a single tab as the next elements of the receiver
crtab: anInteger
Stores a carriage return follwed by anInteger number of tabs as the next elements of the receiver
space
Stores the space charater as the next element of the receiver
tab
Stores the tab character as the next element of the receiver

Stream->ExternalStream
Creation:
Operations:
nextNumber: n
Returns a SmallInteger or LargePositiveInteger representing the next n bytes of the collection accessed by the receiver
nextNumber: n put: v
Stores v, which is a SmallInteger or LargePositiveInteger, as the next n bytes of the collection accessed by the receiver
nextString
Returns a String consisting of the next elements of the collection accessed by the receiver
nextStringPut: aString
Stores aString in the collection accessed by the receiver
padTo: bsize
Skips to the next boundary of bsize characters and returns the number of characters skipped
padTo: bsize put: aCharacter
Skips to the next boundary of bsize characters, writing aCharacter to each character skipped, and returns the number of charcters skipped
padToNextWord
Skip to the next word (even) boundary and returns the number of characters skipped
padToNextWordPut: aCharacter
Skip to the next word (even) boundary, writing aCharacter to each character skipped, and returns the number of characters skipped
skipWords: nWords
Advance position reference nWords
wordPosition
Returns the current position in words
wordPosition: wp
Sets the position reference in words to wp

Index